VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Využití strojové učení v počítačových hrách

Machine Learning in Computer Games

2020 Bc. Marek Horáček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Marek Horáček**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Využití strojové učení v počítačových hrách
Machine Learning in Computer Games

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem diplomové práce je rozšířit simulační systém OtherKosmos o modul samostatně se chovajících postav a jiných entit. Součástí práce je také nastudování a popis technik řízení samostatně se chovajících postav v počítačových hrách.

Body zadání:
1. Popis technik řízení samostatně se chovajících postav (nehratelných postav) v počítačových hrách.
2. Popis možností využití algoritmů z oblasti strojového učení v systémech řízení samostatně se chovajících postav.
3. Porovnání běžně používaných technik pro řízení samostatně se chovajících postav a technik z oblasti strojového učení.
4. Implementace systému samostatně se chovajících postav.

Práce bude obsahovat:
1. Přehled použitých technologií.
2. Implementaci výše popsané funkcionality.
3. Dokumentaci programového řešení s využitím diagramů jazyka UML.

Seznam doporučené odborné literatury:

[1] David M. Bourg and Glenn Seemann. (2004). AI for Game Developers: Creating Intelligent Behavior in Games. 1st Edition. Beijing: O'Reilly. ISBN 9780596005559
[2] Simon Haykin. Neural Networks and Learning Machines: A Comprehensive Foundation. ISBN 0131471392
[3] Suryakumar Balakrishnan Nair, Andreas Oehlke. Learning LibGDX Game Development. 2nd Edition. ISBN 1783554770
[4] ROJAS, Raul. Neural networks: a systematic introduction. New York: Springer-Verlag, c1996. ISBN 3540605053.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2019

Datum odevzdání: 30.04.2020

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 14.5.2020

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 14.5.2020

Na tomto místě bych rád poděkoval vedoucímu mé diplomové práce panu Ing. Davidovi Ježkovi, Ph.D. za odbornou pomoc a celé své rodině za každodenní podporu a motivaci.

Abstrakt

Tato práce se zabývá výzkumem a implementací umělé inteligence v počítačových hrách s ambicí využít v co možná největší míře strojové učení. V rámci práce bylo vytvořeno jednoduché procedurální generování samostatně se chovajících postav, jejich následné učení chůze za pomocí posilovaného učení, vytváření těchto postav v simulačním prostředí, které bylo navíc rozšířeno o systém zkušeností a adaptivní chování postav pohybujících se po tomto prostředí. Výsledkem práce je prezentace zdárného využití strojového učení při implementaci chování samostatně se chovajících postav v simulačním prostředí.

Klíčová slova: umělá inteligence; strojové učení; počítačové hry; diplomová práce

Abstract

This thesis is focused on the research and implementation of artificial intelligence in computer games with the ambition to utilize machine learning in the greatest possible extent. We have created a simple, procedural generation of non-playable characters, their subsequent learning to walk with the help of reinforcement learning, creating such characters in the simulation environment, which was also extended by a system of experiences and adaptive behavior of characters moving in the environment. The result of this thesis is a presentation of a successful utilization of machine learning in the implementation of behavior of non-playable characters in the simulation environment.

Keywords: artificial intelligence; machine learning; computer games; master thesis

Obsah

Seznam použitých zkratek a symbolů

AI	–	Artificial Intelligence
ML	–	Machine Learning
RPG	–	Role-Playing Game
NPC	–	Non-Playable Character
FSM	–	Finite State Machine
GOAP	–	Goal-oriented action planning
SVM	–	Support Vector Machines
DQN	–	Deep Q-networks
MDP	–	Markov decission processes
SQL	–	Structured Query Language
ORM	–	Object Relational Mapping

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Motivací k napsání této práce byla má dlouhodobá záliba v počítačových hrách a současně nadšení pro oblast umělé inteligence (z anglického Artificial intelligence, dále jen AI) a strojového učení (z anglického Machine learning, dále jen ML). V práci jsem se pokusil ukázat, že dnes nejčastěji používané konvenční algoritmy, pomocí kterých se samostatně se chovající postavy (z anglického Non-Playable Characters, dále jen NPC) v počítačových hrách pohybují, chovají a rozhodují, je možné alespoň částečně nahradit algoritmy z oblasti ML a tím přispět k realističnosti a celkovému dojmu z AI těchto postav.

Cílem této práce bylo rozšířit simulační systém The Other Kosmos o modul řešící chování NPC za použití jak algoritmů běžně používaných pro implementace AI v počítačových hrách, tak i algoritmů z oblasti ML.

V první části práce je představen pojem AI z akademického pohledu, jsou uvedeny některé důležité definice AI a je ukázáno základní dělení AI. Další část práce se věnuje popisu běžně používaných technik pro implementaci AI v počítačových hrách. Dále následuje část práce věnující se oblasti ML, jejím modelům a možnostem využití v počítačových hrách. V poslední části je poté představen projekt The Other Kosmos, popsán návrh a implementace AI za použití jak běžně používaných technik pro implementaci AI v počítačových hrách, tak také za použití modelů z oblasti ML, a nakonec jsou zde uvedeny problémy a jejich řešení pomocí implementovaného modulu.

2 Obecně o umělé inteligenci

Oblast AI začala vznikat již po druhé světové válce ve snaze nejen pochopit způsob lidského myšlení, ale také vytvořit nové inteligentní entity. Dnes je umělá inteligence používaná v mnoha oblastech a lze tak na ni pohlížet z mnoha úhlů pohledu. Vzhledem k různosti oborů, ve kterých se o AI pojednává, je velmi složité najít jednu společnou definici, která by zahrnovala všechny aspekty, kterými se AI v různých odvětvích vyznačuje [1]. Na AI lze pohlížet ze čtyř úhlů:

- Myšlení z lidského hlediska – jedná se o automatizaci aktivit spojených s lidským myšlením, jako například rozhodování, řešení problémů a učení.
- Chování po vzoru člověka – vytváření počítačového řešení úloh, v jejichž zvládání jsou dnes lidé lepší.
- Racionální myšlení – formulace výpočtů umožňujících vnímání, uvažování a chování.
- Racionální chování – vytváření inteligentních agentů, přičemž inteligentní agent může být libovolná entita, která je schopna pomocí senzorů z prostředí, ve kterém existuje, získávat informace a na jejich základě poté provádět akce.

Pokud bychom se na AI dívali pouze z hlediska informatiky a matematiky, předešlé definice by plně postačily k vytvoření představy, čím se AI zabývá a co do ni spadá. Mnoho filozofů se však ohradilo vůči těmto čistě technickým pohledům na AI s argumentem, že taková AI je pouze simulací myšlení, nikoli skutečným myšlením [1]. Dnes rozdělujeme AI do dvou hlavních kategorií, silná a slabá AI.

2.1 Silná AI

U silné AI je v systému vyžadována vlastní mysl, hlubší chápání souvislostí, nejen vyřešení nějakého problému. U takové AI předpokládáme schopnost přenášet řešení nějakého problému na řešení dalších problémů.

2.2 Slabá AI

Slabou AI se myslí systémy, které umějí vykonávat nějakou činnost podobně jako by ji prováděl člověk, přičemž se soustředíme pouze na výsledek dané činnosti, aniž bychom brali ohled na hlubší význam inteligence a myšlení. Jedná se o druh AI, která se vždy soustředí na vyřešení úzce definovaného problému a ten řeší velmi dobře. V praxi dnes dominuje využití slabé AI a to napříč obory. Zde je několik příkladů použití AI v praxi:

- Chytří asistenti a chatboti
- Autonomní létání

- Bezpečnost a kontrola obsahu na internetu
- Analýza akciových trhů
- Samořízení a autonomní vozidla
- Analýza medicínských snímků
- Spamové filtry v emailech
- Doporučování videí a hudby na internetu
- Cílení reklamy na internetu

Do oblasti slabé AI patří i AI v počítačových hrách a ML, o kterých pojednávají následující kapitoly [2].

3 AI v počítačových hrách

V počítačových hrách se na AI pohlíží naprosto odlišně než v akademické sféře. Cílem AI ve hrách je vytvořit algoritmy, které zajišťují chování NPC takové, jaké bychom očekávali u lidí a zvířat. Není primárně nutné usilovat o vytvoření inteligence na úrovni srovnatelné s lidskou inteligencí [3]. AI v počítačových hrách můžeme rozdělit na několik druhů:

- **Falešná AI** – tento druh by se dal shrnout větou „Pokud něco vypadá a je cítit jako ryba, pak je to ryba“. Od našich NPC chceme, aby jejich chování vypadalo věrohodně a správně. Vycházíme z vizuální podoby chování, které dobře známe například u reálných bytostí, a to následně promítneme do chování NPC ve hrách. Jedná se o nejsnadnější způsob, jakým lze vytvořit chování NPC. Výsledná AI je dokonalou iluzí inteligence, neexistuje žádná sofistikovaná část řešení. U složitějšího chování se při použití tohoto druhu AI celková architektura systému značně komplikuje a snižuje se rozšířitelnost a přehlednost řešení.

- **Heuristiky** – každá heuristika se skládá z množiny pravidel, které slouží pro řešení mnoha situací, avšak často neobsahuje řešení všech možných situací. Pravidlo může například udávat, jaká akce je v určité situaci nejvíce důležitá, zda se mají důležité akce provádět dříve, než ty méně důležité, případně zda má NPC riskovat anebo zvolit bezpečnější variantu.

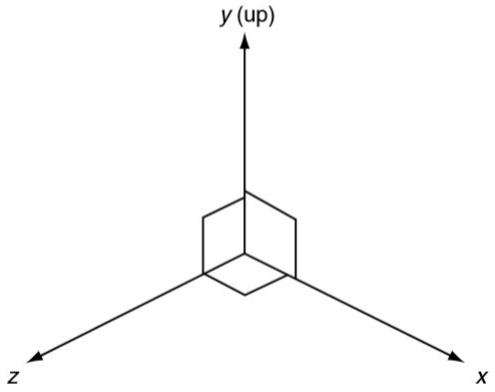
- **Algoritmy** – do této kategorie spadají všechny sofistikované techniky herní AI. Vytváření algoritmů AI pomáhá budovat realistické a zajímavé chování NPC. Falešná AI a heuristiky bývají použity spíše pro doladění chování, které nevypadá příliš realisticky a přirozeně, případně se používají pro určení výjimek v chování, na rozdíl od algoritmů, které slouží pro určení obecných pravidel pro chování NPC a tvoří kvalitní a robustní základ každé herní AI.

Problémy, které AI ve hrách řeší, se dají rozdělit do tří základních oblastí:

- **Pohyb** – jedná se o problémy, které souvisejí s rozhodnutím, jak provést přesun NPC z jednoho místa na druhé. Problém pohybu však není čistě o změně pozice, souvisí také s vhodným načasováním pohybu, trasou pohybu, taktickým rozhodováním, jak a zda se vyhnout ostatním NPC a místům.

- **Rozhodování** – tato část zahrnuje veškeré problémy, u kterých se musí NPC rozhodnout, jaké chování je pro něj v každém momentu hry nejvýhodnější. NPC má většinou na výběr mezi několika akcemi a podle vnitřní logiky systému vybírá jednu nebo více z nich.

- **Strategie** – tato kategorie problémů se týká spíše skupin NPC než jedince. Do této kategorie patří například koordinace skupiny NPC při pohybu, chování skupiny při snaze dosáhnout cíle, ohodnocení dopadu rozhodnutí jedince na zbytek skupiny nebo komunikace a předávání informací v rámci skupiny.



Obrázek 1: Osy 3D herního světa [3]

3.1 Pohyb

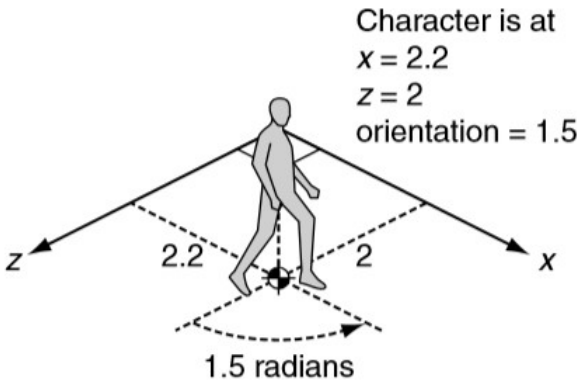
Tato část AI se zabývá pohybem NPC, který může být řešen různými způsoby. S pohybem souvisí také vizuální stránka věci, tedy jak změna polohy NPC na obrazovce vypadá. Je tedy zřejmé, že pohyb jen z části spadá do AI, další velkou složkou je animace pohybu [3].

Nebudeme brát v úvahu různé simulátory, kde pohyb musí být odvozen od fyzikálních vlastností světa, ale budeme se dívat spíše na běžné hry určené pro zábavu. Vstupem většiny algoritmů řešících pohyb je aktuální pozice NPC a cílová pozice, na kterou se chce NPC dostat. Při pohybu se algoritmus musí postarat o vyhýbání překážkám, zamezit narážení do zdí, předcházet kolizím s pohybujícími se objekty, apod. Dalším důležitým vstupem pro tyto algoritmy jsou informace o prostředí, ve kterém se NPC nachází, jako například sklon terénu. Výstupem algoritmů bývá směr, ve kterém se má NPC vydat, někdy také rychlost pohybu, případně koncová pozice.

Pokud je hra vyobrazena ve třech dimenzích (3D), veškeré modely předmětů včetně NPC obsahují tři rozměry. Algoritmy řešící pohyb však s předměty pracují jako s jednorozměrnými prvky, tedy s body na mapě. Kromě reprezentace objektů pro pohyb je také potřeba zmínit, že přestože hra může být 3D, algoritmy pro řešení pohybu typicky pracují pouze se dvěma rozměry. Umístění objektu ve 3D herním světě je složeno ze souřadnic x, y a z (viz obrázek 1). Pozici NPC typicky definujeme pomocí souřadnice x, souřadnice z a tzv. *orientace*, která je uvedena ve stupních nebo radiánech (viz obrázek 2). Změna pozice NPC znamená úpravu hodnoty x a z, případně natočení o určitý počet stupňů nebo radiánů.

Je tedy patrné, že souřadnice y se pro pohyb nepoužívá. Do většiny her se však implementují fyzikální jevy, které známe i z reálného světa. Jedním z těchto jevů je gravitace. Souřadnice y je právě určena pro simulaci gravitace, kde NPC vyskočí, tím zvýší hodnotu souřadnice y, a poté dojde k pádu, tedy postupnému navrácení hodnoty souřadnice y na původní hodnotu.

Samotná změna pozice však není to jediné, co by AI pro pohyb měla řešit. Existuje několik druhů pohybů, kde například změnu pozice ovlivňuje zrychlení objektu, setrvačnost nebo jiné fyzikální veličiny. Dále také záleží, zda je pohyb plynulý nebo skokový, zda se jedná o souhru



Obrázek 2: Určení pozice NPC v herním světě [3]

pohybu více objektů najednou, případně zda je pohyb omezen na rychlosti nebo směru [2]. Podrobnější popis druhů pohybu předmětů ve hrách a definice matematických operací používaných pro pohyb v počítačových hrách je mimo oblast této diplomové práce a nebudeme se tím tedy dále zabývat.

3.2 Vyhledávání cest

Nedílnou součástí pohybu je také vyhledávání cest. NPC se mohou pohybovat různě po prostředí hry. Někdy je cesta pohybu striktně vytyčena vývojářem, jindy může NPC slepě prohledávat prostor, případně se může držet jen určitých míst, které musí navštívit, a jinak se pohybuje nahodile. Vyhledávání cest, někdy taky plánování cest, je použito snad ve všech hrách. Vyhledávání cest se nachází na rozmezí pohybu a rozhodování [3].

Většina her řeší vyhledávání cest pomocí algoritmu nazývaném A*. Je velmi efektivní a jednoduchý na implementaci, bohužel tento algoritmus neumí pracovat přímo se samotným prostředím hry, např. s terénem. Na vstupu vyžaduje vážený graf, který je potřeba z prostředí hry vytvořit. Grafem je myšlena matematická struktura, která je složena z uzlů a hran. Každá hrana vždy spojuje dva uzly, přičemž nikde nespojuje uzel se sebou samým, tedy neexistují smyčky. Uzly grafu představují region, umístění, v herním světě a hrana představuje možný přechod mezi regiony. Vážený graf znamená, že hrany grafu jsou ohodnoceny číslem. Pro vyhledávání cest, ohodnocení hrany grafu představuje cenu přechodu mezi uzly (regiony). Může se jednat například o čas potřebný k provedení přechodu, případně vzdálenost mezi regiony.

Algoritmů řešících problém vyhledávání cest je několik a většina z nich má více variant. Jak již bylo zmíněno dříve, nejčastěji se používá algoritmus A*, avšak dalším známým zástupcem je například Dijkstrův algoritmus. Pro účely této práce nám postačí popsat algoritmus A* (pseudokód algoritmu 3).

A* algoritmus je navržen pro vyhledávání cesty mezi dvěma uzly grafu, přičemž vyhledá jedno možné řešení a to vrátí na výstup. Vstupem algoritmu je vážený graf a dva uzly, ze kterých jeden

představuje počáteční pozici a druhý cílovou pozici. Je potřeba zmínit, že algoritmus nevrací vždy nejlepší řešení, tedy nejkratší cestu v grafu. Při výběru vhodného přechodu na další uzel se algoritmus řídí podle předem určené heuristiky, která přímo ovlivňuje, zda bude vyhledána nejkratší anebo alespoň dostatečně krátká cesta.

Před popsání algoritmu je potřeba vysvětlit, jaký způsobem jsou vybírány přechody mezi uzly. Pro každý navštívený uzel je potřeba určit jeho ohodnocení:

$$f = g + h \tag{1}$$

kde:

f je celková cena uzlu, přesněji se jedná o ohodnocení nejlepší cesty vedoucí přes již navštívený uzel

g je vzdálenost mezi uzlem a počátečním uzlem, přičemž vzdáleností je myšlen součet ohodnocení hran, které byly navštívené při cestě z počátečního vrcholu do daného uzlu

h je heuristika, která odhaduje vzdálenost z aktuálního uzlu do koncového uzlu, přičemž heuristickou funkcí může být například euklidovská vzdálenost mezi dvěma místy na herní mapě

Algoritmus probíhá v následujících krocích [4]:

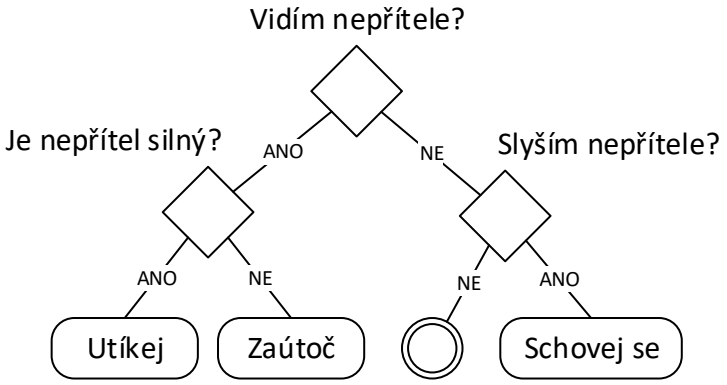
1. Počáteční uzel se přidá do tzv. otevřeného seznamu
2. Opakuje se následující sekvence:
   (a) Vyhledá se nejnižší f hodnota v otevřeném seznamu
   (b) Uzel s nejnižší f hodnotou z předchozího kroku se označí za aktuální uzel a přidá se do tzv. uzavřeného seznamu
   (c) Pro všechny uzly sousední s aktuálním uzlem se provedou následující kroky:
      i. Jestliže přechod na uzel není možné provést (buď hrana směřuje pouze v opačném směru nebo se uzel nachází v uzavřeném seznamu), tak se uzel přidá do uzavřeného seznamu a pokračuje se dalším uzlem sousedním s aktuálním uzlem
      ii. Jestliže uzel není v otevřeném seznamu, tak se do něj přidá. Aktuálním uzlem se nyní stává tento uzel a spočítají se hodnoty f, g a h
      iii. Jestliže je uzel v otevřeném seznamu, zjistí se, jestli je cesta do uzlu kratší pomocí hodnoty g. Jestli je cesta kratší, nastaví se rodič uzlu na aktuální uzel a přepočítají se hodnoty g a f
   (d) Cyklus končí pokud:
      i. Koncový uzel je přidán do uzavřeného seznamu,
      ii. nebo je otevřený seznam prázdný, pak totiž neexistuje cesta z počátečního do koncového uzlu

Algoritmus 1: A*

Result: Sorted list of nodes as a path between the given two nodes

```
1  put the startNode to the openList;
2  while the openList is not empty do
3      currentNode <- the node from openList with lowest f value;
4      remove the currentNode from the openList;
5      add the currentNode to the closedList;
6      if the currentNode equals the goal node then
7          return path
8      foreach child in the currentNodes's neighbors do
9          if the closedList contains the child then
10             continue;
11         compute g, h, f of the child;
12         if the openList contains a position of the child then
13             if child.g is higher than g of a position from the openList then
14                 continue;
15         add the child to the openList;
16     end
17 end
```

Obrázek 3: Pseudokód algoritmu A*



Obrázek 4: Ukázka rozhodovacího stromu

3.3 Rozhodování

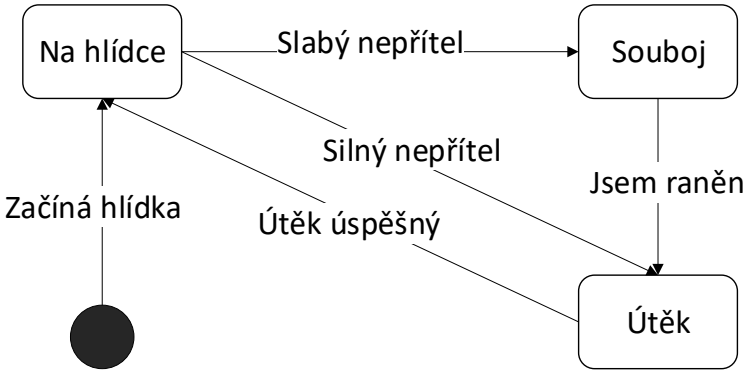
Existuje několik technik pro provádění rozhodnutí. NPC zpracovává sadu informací, na základě kterých vyhodnocuje vhodnou akci, kterou následně provede. Informace mohou být vnější, tedy získané z herního prostředí, anebo vnitřní, což jsou většinou schopnosti a možnosti samotného NPC [3].

3.3.1 Rozhodovací strom

Rozhodovací stromy jsou rychlé, jednoduché na implementaci a snadné na pochopení. Jedná se o nejjednodušší techniku provádění rozhodnutí, avšak jsou velmi užitečné a při použití různých rozšíření se z nich stává poměrně sofistikovaný nástroj.

Vstupem algoritmu je sada informací a ta je mapována na výstup v podobě možných akcí. Rozhodovací strom je vytvořen z množiny propojených rozhodovacích uzlů. Počáteční uzel stromu je nazýván kořenem stromu. Každé rozhodnutí, počínaje kořenem, posouvá proces rozhodování do jednoho z potomků zpracovaného uzlu. Každé rozhodnutí je prováděno na základě znalosti NPC. Proces rozhodování končí ve chvíli, kdy aktuálně prováděný uzel nemá žádného potomka, přičemž takový uzel je nazýván listovým uzlem. Každý listový uzel stromu má přiřazenou akci, která je v případě ukončení rozhodování v tomto uzlu provedena [3].

Na obrázku 4 je znázorněn příklad takového rozhodovacího stromu. Kořenem stromu je rozhodnutí, zda NPC vidí svého nepřítele. Pokud je odpověď kladná, rozhodování pokračuje levou větví do uzlu s rozhodnutím, zda je nepřítel silný. Pokud je odpověď opět kladná, NPC provede akci *Utíkej*. Podobně jsou uzly zpracovány i v ostatních případech. Pokud NPC nepřítele nevidí a současně jej neslyší, pak není provedena žádná akce, což je v rozhodovacím stromu naznačeno dvojitým kolečkem. Rozhodovací uzly mohou obsahovat podmínky různých datových typů:

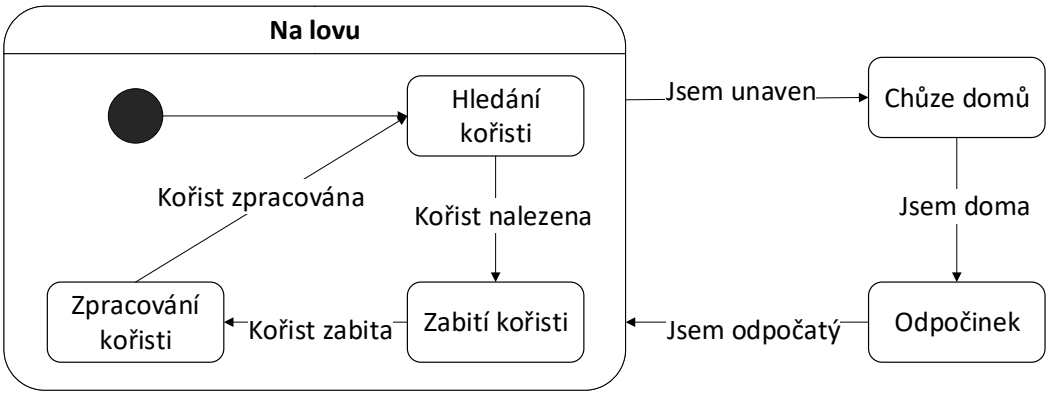


Obrázek 5: Ukázka konečného stavového automatu

- Logická hodnota – podmínka je typu pravda/nepravda a uzel obsahuje 2 potomky, přičemž první je pro logickou hodnotu pravda a druhý pro logickou hodnotu nepravda.
- Výčet hodnot – uzel vybírá dle hodnoty takového potomka, který je určen pro zpracování v případě konkrétní výčtové hodnoty. Může existovat i výchozí potomek, pokud žádná z výčtových hodnot se nerovná aktuálně rozhodované hodnotě.
- Číselná hodnota – u tohoto typu podmínky mají potomci uzlu určené rozsahy hodnot, pro které mají být vybrány, pokud hodnota spadá do rozsahu hodnot daného potomka.
- 3D vektor – vyhodnocení podmínky probíhá na základě velikosti vektoru, kde každý potomek má určen rozsah velikostí, pro které je uzel vybrán. Příkladem použití může být například vzdálenost NPC od hráče, kde rozhodnutí probíhá na základě velikosti vektoru rozdílu pozic NPC a hráče.

3.3.2 Konečný stavový automat

Konečný stavový automat (z anglického Finite State Machine, dále jen FSM) je nejpoužívanější technikou pro provádění rozhodnutí v počítačových hrách. NPC se nachází vždy v jednom stavu z více možných. Chování NPC je definované pro každý stav zvlášť a dokud se stav nezmění, NPC stále provádí akce určené tímto stavem. Stavy jsou navzájem propojené přechody. Každý přechod vede z jednoho stavu do druhého na základě asociačních pravidel. Pokud systém odhalí, že podmínky pro přechod mezi stavy byly splněny, FSM změní svůj stav pomocí přechodu do nového stavu. V každé iteraci herního cyklu probíhá aktualizace FSM, při které dochází k vyhodnocení přechodových podmínek, případné změně stavu a následně k vykonání chování definovaném v aktuálním stavu FSM. Díky dekompozici chování NPC do jednotlivých stavů se kód definující AI NPC stává daleko přehlednějším, jednodušším na údržbu a odolnějším vůči chybám [3].



Obrázek 6: Ukázka hierarchického konečného stavového automatu

Příklad FSM je vidět na obrázku 5. Počáteční stav, který je zobrazený černým kolečkem, je výchozím stavem při vytvoření NPC. Když dojde ke splnění podmínky *Začíná hlídka*, FSM se přepne do stavu *Na hlídce*, ve kterém setrvá až do doby, kdy NPC potká nepřítele, který je buď silný anebo slabý, přičemž se FSM přepne do stavu *Souboj*, resp. *Útěk*. Pokud je FSM ve stavu *Souboj*, setrvá v něm až do té doby, než dojde ke splnění podmínky *Jsem raněn*. V takovém případě se FSM přepne do stavu *Útěk*, ve kterém setrvá až do doby, než je splněna podmínka *Útěk úspěšný*.

Jednou z variant FSM je hierarchický FSM, jehož hlavní myšlenka spočívá v tom, že každý stav FSM může mimo vlastního chování obsahovat také vnitřní FSM, který ještě více rozděluje chování pro daný stav. Nutno podotknout, že v hierarchickém FSM můžeme mít dva druhy přechodových podmínek. Jeden typ určuje podmínky přechodu mezi jedním stavem do druhého, druhý typ představuje přechod ze všech vnitřních stavů daného FSM do jiného stavu. U druhého typu je přechod zajištěn bez ohledu na to, v jakém z vnitřních stavů se FSM nachází, a je tedy platný pro všechny vnitřní stavy.

Ukázka hierarchického FSM je znázorněna na obrázku 6. FSM obsahuje stavy *Na lovu*, *Chůze domů* a *Odpočinek*. Stav *Na lovu* navíc obsahuje vnitřní FSM, který se skládá ze stavů *Hledání kořisti*, *Zabití kořisti* a *Zpracování kořisti*. Způsob přecházení mezi jednotlivými stavy je shodný se způsobem popsaným u předchozího příkladu. Navíc nám však tady přibyl počáteční stav uvnitř vnitřního FSM, který neobsahuje žádnou přechodovou podmínku, a je z něj zřejmé, že kdykoli dojde ke změně stavu na stav *Na lovu*, je vždy vnitřní FSM tohoto stavu nastaven na stav *Hledání kořisti*. Dále je zde rozdílná přechodová podmínka *Jsem unaven*, která je odlišná v tom, že je platná pro všechny vnitřní stavy vnitřního FSM stavu *Na lovu*.

3.3.3 Behaviorální strom

Behaviorální stromy mají mnoho společného s hierarchickými FSM, hlavní složkou však není stav ale úloha. Dílčí úlohy jsou rozděleny do podstromů a dohromady tvoří složitější akce. Každá

úloha má jednotné rozhraní a je tedy možné s nimi pracovat obecně, bez nutnosti znát jejich detailní implementaci. Každá úloha má přiřazen určitý procesorový čas na provedení vnitřní logiky a následně vrací kód, který indikuje buďto úspěch, selhání, potřebu pokračovat v provádění, případně jiné informace, na které je potřeba reagovat. Jednotlivé úlohy by měly být dostatečně jednoduché a pokud dojde k jejich nadměrnému rozšíření, měly by se rozdělit na menší podúlohy [3]. Typy úloh:

- **Podmínka** – podmínky testují nějakou vlastnost hry. Může se jednat o testování stavu NPC, polohu nějakého předmětu, případně jiné vlastnosti herního světa. Každé testování by mělo tvořit samostatnou podmínku. Podmínka opět vrací kód, který indikuje úspěch nebo selhání.

- **Akce** – akce upravují stav hry. Mohou existovat stavy pracující s animacemi, pohybem NPC, změnou vnitřního stavu NPC anebo jinou částí hry.

Primární rozdíl mezi rozhodovacími stromy a behaviorálními stromy je ten, že u behaviorální stromů existuje jednotné rozhraní pro všechny uzly. Jednotlivé uzly tedy vůbec nemusejí znát nic o zbytku stromu. Podmínky i akce se musejí nacházet vždy v listových uzlech stromu. Nelistové uzly se nazývají kompozitní uzly a jejich návratový kód přímo určují návratové hodnoty jejich potomků. Typy kompozitních uzlů:

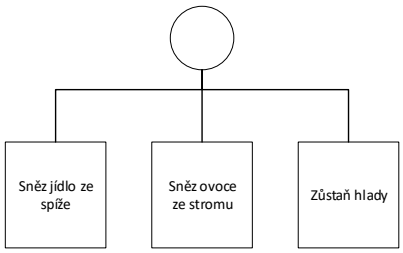
- **Selektor** – vrací okamžitě kód indikující úspěch, pokud jeden z potomků byl proveden úspěšně. Pokud libovolný potomek selže, pokračuje se na dalšího potomka, dokud není zpracován poslední potomek, poté vrací kód o neúspěchu. Příklad selektoru je na obrázku 7. V tomto příkladu máme jako první akci *Sněz jídlo ze spíže*, dále pokud její provedení selže, tak přichází na řadu akce *Sněz ovoce ze stromu*, která pokud také selže, provede se akce *Zůstaň hlady*.

- **Sekvence** – reprezentuje sérii úloh, kde pokud dojde k selhání alespoň jedné z nich, selže celá větev. Pouze pokud by došlo k úspěšnému provedení všech potomků, větev vrací kód úspěchu. Ukázka sekvence je na obrázku 8, kde máme jako první podmínku *Je medvěd blízko?*, dále akci *Otoč se*, která se provede, pokud je první podmínka splněna, a nakonec akci *Utíkej*, která je provedena v případě, že předchozí akce je dokončena úspěšně.

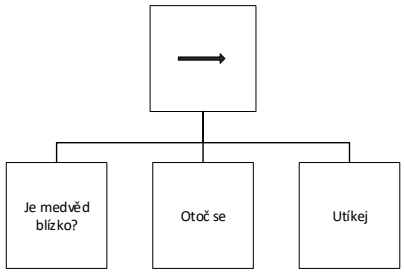
Speciálním případem uzlu je tzv. dekorátor. Tento uzel má vždy pouze jednoho potomka a jeho úkolem je upravovat vstup nebo výstup potomka. Jednoduchým příkladem může být dekorátor, který provádí inverzi návratového kódu, tedy úspěch mění na selhání a naopak.

3.3.4 Cílově orientované chování

Jedná se o obecný pojem, který zahrnuje veškeré techniky, které vyhodnocují kroky potřebné k dosažení určitých cílů. Předchozí algoritmy byly výhradně konstruovány napevno s téměř žádným nebo jen malým prostorem pro plánování a adaptování se na aktuální podmínky. Často



Obrázek 7: Ukázka selektoru behaviorálního stromu



Obrázek 8: Ukázka sekvence behaviorálního stromu

se díky tomu může stát, že se hráč po určitém čase začne všímat systému v chování NPC a hra pro něj začne být předvídatelná. Cílově orientované chování přináší jisté zmírnění tohoto efektu a jeho použitím se stává chování NPC o něco více různorodé [3].

Ve hře může existovat mnoho cílů, někdy se jim též říká motivy, přičemž některé jsou pro NPC aktivní, tedy snaží se jich dosáhnout, a jiné neaktivní. Každý cíl má určenou míru důležitosti, která je reprezentovaná číslem. Cíle s vyšší mírou důležitosti více ovlivňují chování NPC. Kromě cílů je k dispozici i množina akcí. Akce mohou být generovány z jednoho centrálního místa v systému, ale mohou být také vytvářeny různými dalšími objekty světa. Množina dostupných akcí je vždy závislá na konkrétním stavu hry. Akce, kterou je zajištěno splnění cíle, často nemusí být k dispozici v konkrétním stavu hry. V takovém případě je potřeba provést jednu, případně více jiných akcí, které NPC k cíli pouze přiblíží.

Nalezené vhodné strategie, tedy sekvence akcí, která vede k dosažení cíle, nazýváme plánováním. Plánovací techniky jsou různé, většina ale pracuje s množinou dostupných akcí jako se stromem, kde uzly představují stavy hry a hrany přechodové akce.

Konkrétním algoritmem cílově orientovaného chování je cílově orientované akční plánování (z anglického Goal-oriented action planning, dále jen GOAP). Pokud si NPC definuje cíl, algoritmus začne prohledávat stavový prostor, vybere plán s nejnižší cenou a ten poté předá NPC k provedení. Každá dostupná akce v daném stavu hry obsahuje předpoklady, účinky a cenu. Předpoklady můžeme chápat jako stav, který je vyžadován pro provedení akce. Účinkem je změna stavu, která je provedena při vykonání akce. Vezměme si následující příklad:

Množina stavů: *MášDřevo*, *DřevoJeNaštípáno*, *JeTiTeplo*
Množina akcí:

- *Zapal dřevěné třísky*
  - Předpoklady: *DřevoJeNaštípáno*
  - Účinky: *JeTiTeplo*
  - Cena: 5
- *Naštípej dřevo*
  - Předpoklady: *MášDřevo*
  - Účinky: *DřevoJeNaštípáno*
  - Cena: 30
- *Počkej na léto*
  - Předpoklady:
  - Účinky: *JeTiTeplo*
  - Cena: 100
- *Najdi dřevo*
  - Předpoklady: *!MášDřevo*
  - Účinky: *MášDřevo*
  - Cena: 15

Aktuální stav světa je následující: *!MášDřevo*, *!DřevoJeNaštípáno*, *!JeTiTeplo*. Našim cílem je dosáhnout stavu: *JeTiTeplo*. Existují 2 sekvence akcí, které vedou k dosažení cíle:

- *Najdi dřevo -> Naštípej dřevo -> Zapal dřevěné třísky* (výsledná cena je 50)
- *Počkej na léto* (výsledná cena je 100)

Z množiny možných sekvencí akcí by algoritmus vybral první možnost, jelikož se jedná o levnější volbu.

3.4 Strategie

Poslední kapitolou herní AI je strategie. Techniky pro rozhodování mají dvě důležitá omezení, a sice že jsou odkázány na použití právě jedním NPC a nedokáží být spojeny s rozhodováním ostatních NPC a vytvářet rozhodnutí na základě globální situace [3].

Prvním příkladem použití strategie je vyhledávání cest. V případě A* algoritmu jsme měli prostředí hry reprezentované grafem složeném z uzlů, které představují určité regiony (místa) ve

hře, a hran, které představují propojení těchto míst. Pokud bychom chtěli vytvořit strategické vyhledávání cest, museli bychom přidat do uzlů informaci o taktické situaci v uzlu. Kromě ocenění hran by tedy vyhledávání cest muselo brát v potaz i tyto taktické informace. Příkladem může být míra bezpečí na daném místě. Pokud by se NPC vydalo cestou velmi rychlou a jednoduchou, ale poměrně nebezpečnou, mohlo by často dojít do cíle s problémy, případně do cíle vůbec nedorazit.

Dalším zástupcem strategické AI je taktická analýza. Taktickou analýzou se hlavně myslí modelování a předpovídání dopadu rozhodnutí, míra ovlivnění stavu hry určitou akcí případně vedlejší účinky chování určitého chování hráče nebo NPC. Pokud by například existovala hra, ve které by hráč hrál proti počítači a navzájem se snažili pomocí svého vojska porazit nepřítele, bylo by vhodné použít strategii pro rozmístění vojsk po mapě, aby došlo k poražení nepřítele. Dále může být použita taktická analýza při vyhodnocování struktury terénu hry, kde NPC se musí rozhodnout, jakým způsobem jeho chůzi ovlivní sklon povrchu, nutnost obcházení překážek, případně zpomalení v následku potřeby překonat řeku na své cestě.

Poslední podkategorií strategické AI jsou koordinované akce. Ve spoustě strategických hrách může hráč ovládat více NPC současně, většinou pomocí rozdávání příkazů. NPC, které dostane od hráče příkaz, začne konat takovým způsobem, aby hráčův příkaz splnilo co možná nejlépe. Jsou však také strategické hry, ve kterých hráč pouze vyjádří záměr dosáhnout určitého cíle, například postavení budovy, a nepřiřazuje NPC jednotlivé úkoly, které musejí být splněny k tomu, aby byla budova zdárně postavena. V takovém případě musejí NPC začít konat samostatně, ale současně také koordinovaně s ostatními, aby došlo ke splnění úkolu co možné v nejlepším čase a kvalitě. Ke komunikaci mezi NPC slouží různé systémy zpráv, které zajišťují spolupráci mezi NPC.

O strategické AI je toho možné napsat daleko více, avšak v rámci této diplomové práce není potřeba zacházet příliš do detailu, jelikož v implementaci nebyla strategická AI použita.

4 Strojové učení

ML je oblast počítačové vědy, která se zabývá učením počítačů z dat. Arthur Samuel definoval strojové učení jako obor, který dává počítačům schopnost se učit bez potřeby explicitního naprogramování. Další známou, více technickou, definicí je, že se jedná o počítačový program, který má za úkol učit se ze zkušenosti E s danou úlohou T a hodnotící funkcí P, pokud se hodnocení na úloze T, měřeném funkcí P, zlepšuje se získanou zkušeností E. ML bylo poprvé široce použito po roce 1990 ve spamových filtrech [5]. Využitím předchozí definice můžeme tento příklad použití popsat následovně:

- Spamový filtr využívající ML je program schopný se naučit označovat emaily za spam (nevyžádaná pošta) nebo za ham (vyžádaná pošta) z příkladů emailů, který jsou již označeny uživateli za spam nebo ham.
- Tyto příklady, ze kterých se systém učí, se nazývají trénovací sada, přičemž každý jeden příklad z trénovací sady se nazývá trénovací instance.
- Úloha T je označovat nové emaily za spam, zkušenost E je trénovací sada a hodnotící funkce P může být například poměr správně a chybně označených emailů za spam.
- Hodnotící funkce P je v oblasti ML obecně nazývána jako přesnost.

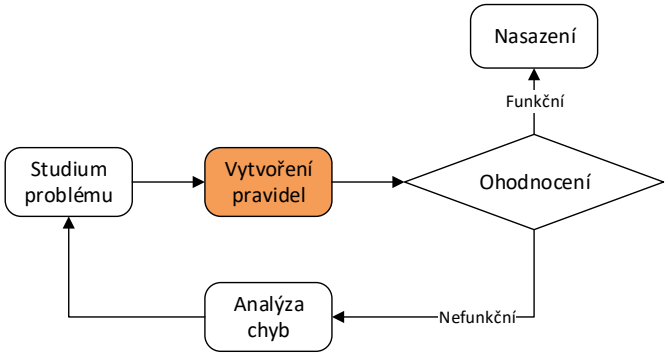
4.1 Kdy použít ML

Pro pochopení smyslu použití ML a vysvětlení jeho výhod zůstaneme u příkladu spamového filtru a popíšeme si jeho řešení tradičními algoritmy [5]:

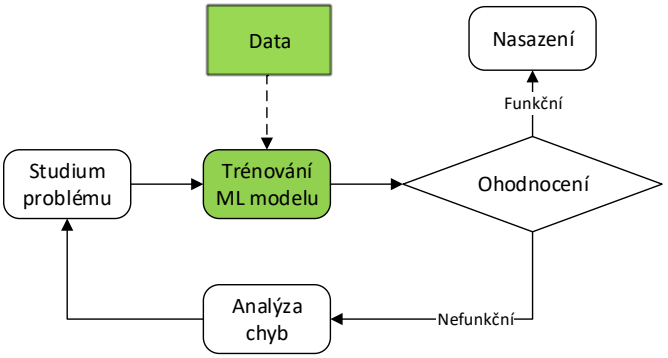
1. V prvním kroku je třeba zjistit, co mají spamy společné, která slova a fráze se v nich vyskytují (například slova: 4U, credit card, free, amazing).
2. Když zjistíme společné rysy spamů, jsme schopni vytvořit detekční algoritmus, který pro každý zjištěný vzor a frázi ověří jeho existenci v emailu a poté označí email za spam nebo ham.
3. Poté je potřeba program otestovat a ověřit, zda funguje dostatečně a v případě potřeby opakovat kroky *1* a *2*, dokud nebude algoritmus obsahovat taková pravidla, která dostatečně dobře odhalí spam. Na obrázku 9 je ukázka takového programu.

Pokud by náš problém nebyl příliš jednoduchý, algoritmus řešící tento problém by pravděpodobně obsahovat dlouhý seznam komplexních, ručně psaných, pravidel.

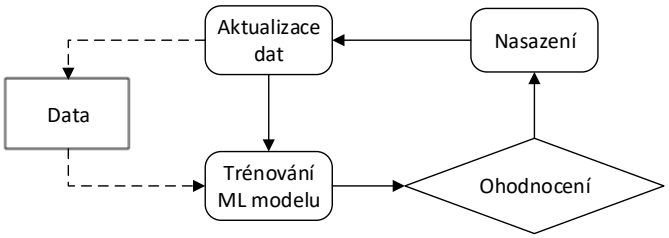
Na rozdíl od řešení pomocí tradičních algoritmů, spamový filtr postavený na principech ML je schopen automaticky se naučit a rozhodnout, která slova a fráze jsou vhodnými predikáty pro detekci spamu. Výsledný program je o dost kratší a snadný na údržbu, často navíc i o něco přesnější. Příklad takového programu je na obrázku 10.



Obrázek 9: Tradiční spamový filtr



Obrázek 10: Spamový filtr využívající ML



Obrázek 11: Spamový filtr s detekcí změn využívající ML

Navíc je potřeba u problému spamových filtrů brát v potaz fakt, že lidé, kteří spamy posílají, často mění strukturu těchto emailů a snaží se spamové filtry obcházet. Například pokud náš spamový filtr má mezi pravidly pro detekci spamu slovo „4U“, je možné toto slovo změnit na slovo „For U“. Tradiční algoritmus by si s takovou změnou neporadil a email by chybně označil za ham. Programátoři by museli pravidla neustále aktualizovat a ručně přepisovat. Na rozdíl od tradičního spamového filtru, spamový filtr využívající ML automaticky rozpozná, že slovní spojení „For U“ se nyní vyskytuje příliš často v emailech označovaných uživateli jako spam, a proto by začal opět emaily s touto frází označovat za spamy. Na obrázku 11 je znázorněn tento spamový filtr s detekcí změn.

Další oblastí pro využití ML jsou problémy, které jsou příliš složité pro řešení tradiční cestou, případně pro ně není znám žádný algoritmus. Příkladem může být rozpoznání mluveného hlasu ve zvukovém záznamu. Bylo by zapotřebí vytvořit program, který by rozpoznával co nejvíce slov, zaznamenat přesně tón a intenzitu zvuku. Řešení pomocí ML by bylo možné provést tím způsobem, že by se algoritmus učil z příkladů slov a následně by tato slova hledal v přiloženém zvuku. ML je také možné využít u dolování dat, při nímž dochází k hledání vzorů ve velkém množství dat, které by člověk jen velmi složitě ručně procházel. V takovém případě ML je schopno pomoci člověku pochopit význam, vazby a vzory v datech. ML je tedy vhodné použít ve čtyřech hlavních případech:

1. Problém má řešení, ale to se skládá z velkého množství ručně psaných pravidel, která jsou složitá na údržbu. V takovém případě může ML pomoci řešení zjednodušit a problém vyřešit lépe.
2. Problém je příliš komplexní a neexistuje žádné vhodné řešení.
3. Měnící se prostředí problému, ve kterém je ML schopno se adaptovat na nová data.
4. Při získávání informací z velkého množství dat.

Techniky ML je možné rozdělit do několika kategorií z různých pohledů. Následující část práce se věnuje právě způsobům rozdělení technik ML.

4.2 Dělení modelů ML

Modely ML mohou být z pohledu, zda existuje nějaký učitel, rozděleny do čtyř kategorií, a sice do kategorie učení s učitelem, učení bez učitele, částečné učení s učitelem a posilované učení [5].

4.2.1 Učení s učitelem

Jedná se o způsob učení, při kterém existuje množina trénovacích dat, kde u každé instance je uvedeno označení, na základě kterého se algoritmus učí při práci s touto trénovací množinou. Typickou úlohou učení s učitelem je klasifikace. Spamový filtr je dobrý příklad. Algoritmus je v tomto případě naučen díky velkému množství příkladů emailů, u kterých je vždy uvedeno, zda se jedná o spam nebo ham (třída instance). Tento algoritmus se tímto způsobem naučí klasifikovat další nové, předem neznámé, emaily.

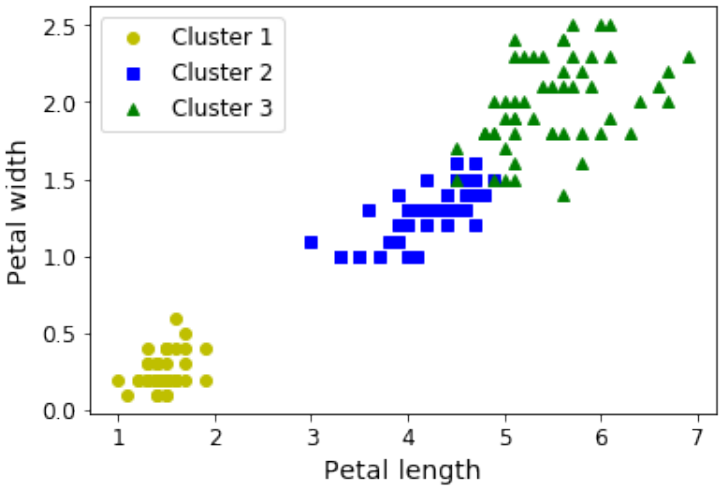
Další úlohou učení s učitelem je odhadování numerické hodnoty pro konkrétní sadu vlastností, které se nazývají predikáty. Tato úloha se nazývá regrese. Příkladem může být program na odhadování ceny auta. Vstupní trénovací sada obsahuje mnoho příkladů aut, kde pro každé auto je uvedena sada vlastností (například počet ujetých kilometrů, stáří a značka vozu) a skutečná cena auta. Systém se naučí na základě příkladů odhadovat cenu dalších, dříve neznámých, aut [5]. Příklady algoritmů:

- K-nejbližších sousedů
- Lineární regrese
- Logistická regrese
- Support Vector Machines (dále jen SVM)
- Rozhodovací stromy a náhodné lesy
- Neuronové sítě

4.2.2 Učení bez učitele

Odvětví ML bez učitele pracuje s množinou trénovacích dat, která nemají určeno označení (třídu). Nejrozšířenější skupina algoritmů ML bez učitele se nazývá shlukování (znázorněno na obrázku 12). Příkladem algoritmu pro shlukování je k-průměrů (z anglického k-Means) nebo hierarchická shluková analýza (z anglického Hierarchical Cluster Analysis). Další skupina algoritmů ML bez učitele jsou algoritmy určené pro redukci dimenze, například Principal Component Analysis (PCA) [5].

Pokud bychom se zaměřili na shlukové algoritmy, mohli bychom si představit problém, kde máme mnoho dat o určité skupině osob a chtěli bychom najít spojitost mezi některými z nich. Shlukový algoritmus by začal vyhledávat spojení mezi osobami a na základě těchto spojení



Obrázek 12: Příklad shlukování

by rozdělil celkovou množinu lidí do shluků. V případě použití hierarchického shlukování by algoritmus provedl rozdělení množiny lidí do shluků na několika úrovních.

Další úloha pro ML bez učitele, redukce dimenze, se týká problému, kdy máme data příliš komplexní a nejsme schopni z nich vyčíst dostatečné informace. Cílem redukce dimenze je tedy zjednodušit data, ale současně se snažit o minimalizace ztráty informace. Častým důvodem pro redukci dimenze je například potřeba vizualizace dat, jelikož pokud by data měla více než tři dimenze, už by se velmi obtížně vizualizovala a člověk by z pohledu na ně příliš nevyčetl. Jedním ze způsobů provedení redukce dimenze je provedení sjednocení atributů, která mají vysoký korelační koeficient, přičemž tento postup se nazývá výběr vlastností (z anglického Feature selection).

4.2.3 Částečné učení s učitelem

Jedná se o skupinu algoritmů pracujících s částečně označenými daty, typicky s velkou množinou neoznačených dat a malou množinou označených dat. Příkladem mohou být služby pro ukládání fotografií na internetu, přičemž tyto služby automaticky rozpoznávají, na kterých fotografiích se nacházejí stejní lidé. Pokud tedy algoritmus umí rozpoznat obličej člověka, je schopen rozdělit fotografie podle toho, zda se na nich nachází stejná osoba či nikoliv [5].

4.2.4 Posilované učení

O dost rozdílný okruh ML je posilované učení. Proces učení probíhá pomocí pozorování prostředí tzv. agentem, výběrem a prováděním akcí, a následném získání odměny nebo penále v závislosti na tom, zda měla akce pozitivní nebo negativní důsledky. Algoritmus se musí naučit sám, jakou

strategii, politiku, je nejvýhodnější používat v daném čase. Politika určuje, jakou akci má agent vybrat v dané situaci. Příkladem algoritmu s posilovaným učením je Q-Learning [5].

Další rozdělení ML je v závislosti na tom, zda je algoritmus schopen se učit inkrementálně z nově získaných dat.

4.2.5 Dávkové učení

Tato skupina algoritmů není schopna se učit inkrementálně, musejí se vždy přeučit z celé trénovací datové sady. Přeučení vždy trvá déle než u inkrementálního způsobu učení. Algoritmus je vždy naučen předem a využívá se v produkci bez jakýchkoli změn. Až když dojde k potřebě využít nově získaná dat, algoritmus se přeučí a znovu uvede do produkce. Tento způsob se také nazývá offline učení. Celý proces přeučování se pochopitelně dá velmi dobře automatizovat.

4.2.6 Online učení

S tímto typem učení je algoritmus schopen přijímat nové trénovací instance, buďto jednotlivě nebo v malých dávkách, a následně provést přeučení, které je v tomto případě daleko rychlejší, jelikož dochází ke zpracování pouze malé datové sady. Důležitým parametrem online učení, který udává, jak rychle se algoritmus adaptuje na změněná data, se nazývá koeficient učení (z anglického learning rate). Pokud je koeficient učení příliš vysoký, algoritmus je rapidně ovlivněn novými daty, ale rychle stará data zapomíná. Naopak pokud je koeficient příliš nízký, tak má algoritmus větší setrvačnost, tedy přeučuje se velmi pomalu. Volba koeficientu učení je jedním z klíčových faktorů ovlivňujících kvalitu algoritmu ML.

Dalším rozdělením algoritmů ML je podle způsobu generalizace. Většina úloh pro ML souvisí s vytvářením predikcí. To znamená, že máme určitý počet trénovacích příkladů, pomocí kterých algoritmus naučíme řešit náš problém, ale současně chceme, aby byl algoritmus schopen problém řešit i pro příklady, které ještě nikdy neviděl. Tento princip se nazývá generalizace, přičemž existují 2 přístupy: na základě instance (instance-based learning) a na základě modelu (model-based learning) [5].

4.2.7 Instance-based learning

Tento způsob generalizace řeší úlohu pro nové instance na základě podobnosti s již naučenými instancemi. Je vyžadováno měření podobnosti mezi instancemi. Pokud algoritmus získá na vstupu, do té doby nepoznanou, datovou instanci, pokusí se mezi známými instancemi najít nejpodobnějšího zástupce, někdy i více než jednoho, pomocí naměřené podobnosti a vyřešit úlohu pro novou instanci po vzoru nejpodobnějšího zástupce. Měření podobnosti může být provedeno například pomocí kosinové podobnosti nebo Euklidovské vzdálenosti.

4.2.8 Model-based learning

Druhým způsobem, jak provádět generalizaci, je vytvořit model, který odpovídá trénovací datové sadě, a na jeho základě provádět predikce. Pokud bychom měli problém klasifikace ve dvourozměrném prostředí, modelem by byla křivka, případně více křivek, které vhodně oddělují jednotlivé instance dle třídy, do které patří.

Pokud zvolíme generalizaci pomocí modelu, je potřeba nejprve tento model vytvořit. Tento proces je nazýván výběrem modelu (z anglického model selection). Každý model má většinou několik parametrů, se kterými se dá manipulovat a tím model více či méně odpovídá trénovací datové sadě [5]. Více se zaměříme na výběr modelu a ladění parametrů v kapitolách 4.3.7 a 4.4.

4.3 Postup použití ML

V této kapitole si popíšeme, jaké kroky je potřeba provést pro vytvoření systému využívajícího ML, jak je potřeba si definovat problém, který má ML řešit, jaké zpracování dat musí předcházet samotnému ML a jakým způsobem zvolit vhodný model a naučit jej řešit problém.

4.3.1 Definice problému

Na začátku je vždy potřeba zjistit, co nám má model ML přinést, jaký problém nám má vyřešit, jak je možné ohodnotit model, zda funguje správně, a kolik času máme na samotné učení. Jednou ze základních otázek ohledně problému je, zda se jedná o problém řešitelný ML z kategorie s učitelem, bez učitele nebo posilovaného učení. Další otázku je, zda je našim úkolem provádět klasifikaci, regresi, případně jinou úlohu. Dále zda je možné využít dávkového učení anebo online učení.

4.3.2 Výběr hodnocení chyby

Dalším krokem je zjištění, jakým způsobem se bude určovat, jak dobře model funguje. Pro regresi se často používá metrika RMSE (z anglického Root Mean Square Error). Tato metrika měří směrodatnou odchylku chybovosti predikce modelu. Rovnice pro výpočet chyby vypadá následovně:

$$\mathrm{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2} \tag{2}$$

Další metrikou, kterou můžeme pro měření chyby použít je MAE (z anglického Mean Absolute Error). Tato metrika je lepší použít v případě, kdy trénovací data obsahují příliš odlehlých pozorování. Rovnice vypadá následovně:

$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right| \tag{3}$$

4.3.3 Vytvoření testovací datové sady

Další nezbytnou součástí návrhu řešení pomocí ML je vytvoření testovací datové sady z původní datové sady. Nejjednodušším způsobem, jak testovací sadu vytvořit, je náhodně vybrat určitou část, typicky 20 %, z původní datové sady. U náhodného výběru podmnožiny datové sady hrozí výběr nereprezentativního vzorku dat. Pro příklad si můžeme vzít datovou sadu osob, která se skládá z 51,3 % žen a 48,7 % mužů. Pokud bychom prováděli čistě náhodný výběr, mohl by se rapidně změnit poměr mužů a žen ve výsledné datové sadě. Lepší způsob je provádět výběr tak, aby i výsledný vzorek měl stejný poměr mužů a žen. Takový výběr se nazývá vrstvený výběr (z anglického stratified sampling) [5].

S vytvářením testovací datové sady dále souvisí pojem křížové ověření (z anglického cross-validation). Jak již název napovídá, testovací datovou sadu vytváříme z toho důvodu, abychom výsledný model otestovali. Vhodné je však testovat model i během trénování, případně během návrhu modelu, a sice proto, abychom zjistili, zda model kromě trénovací datové sady správně ohodnocuje i dříve neznámé instance, tedy jak dobře generalizuje problém. Za tímto účelem vytváříme pomocí křížového ověření validační datovou sadu, na které se poté ověřuje, zda model není přeučený, tedy že funguje velmi dobře na trénovací datové sadě, ale o dost hůře na validační datové sadě, anebo zda není nedoučený, tedy jestli model funguje ne příliš dobře i na trénovací datové sadě. Křížené ověření rozděluje původní datovou sadu na několik trénovacích a testovacích datových sad, na kterých se poté daleko lépe pozná chybovost daného modelu, jelikož při výběru různých vzorků dat je i naměřená chyba různá. Až při opakovaném měření chyby na různých testovacích datových sadách zjišťujeme přesnější chybu modelu. Křížové ověření se také velmi hodí pro nastavení tzv. hyper-parametrů, tedy parametrů, které nesouvisejí přímo s modelem, ale ovlivňují průběh učení [6]. Takovým parametrem je například koeficient učení, který byl popsán v kapitole 4.2.6.

4.3.4 Korelace

Jednou z částí předzpracování datové sady je odstranění, případně transformace, atributů, které spolu vysoce korelují. Pro datovou sadu se spočítá tzv. standardní korelační koeficient mezi všemi dvojicemi atributů. Korelační koeficient je v rozsahu od -1 do 1, přičemž číslo blízké 1 znamená, že se jedná o velmi silnou pozitivní korelaci. To znamená, že pokud roste hodnota jednoho atributu, hodnota druhého atributu také roste. Pokud je koeficient blízký -1, pak se jedná o velmi silnou negativní korelaci. Tedy pokud hodnota jednoho atributu roste, hodnota druhého atributu zákonitě klesá. Pokud je koeficient blízký nule, pak spolu atributy nekorelují. Atributy s vysokou korelací mohou zhoršovat schopnost modelu se naučit, na tyto atributy se dá pohlížet jako na jeden atribut s větší důležitostí, než mají ostatní atributy. Někdy je vhodné jeden z atributů s vysokou korelací odstranit, jindy provést vytvoření nového atributu, který

vznikne z původních dvou atributů s vysokou korelací [5]. Korelaci lze spočítat následovně [6]:

$$\mathrm{Cor}(X,Y) = \frac{\sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{m}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{m}(y_i - \bar{y})^2}} \tag{4}$$

kde:

x_i je i-tá hodnota atributu x

$\bar{x}$ je průměrná hodnota atributu x

y_i je i-tá hodnota atributu y

$\bar{y}$ je průměrná hodnota atributu y

m je počet instancí datové sady

4.3.5 Čištění dat

Většina algoritmů ML neumí pracovat s chybějícími atributy, což je potřeba vyřešit již ve fázi předzpracování dat. Pokud narazíme na chybějící hodnotu atributu, můžeme buď instanci odstranit nebo odstranit atribut ze všech instancí, případně hodnotu atributu v dané instanci nahradit mediánem nebo průměrem hodnot atributu.

Dále se mohou mezi daty vyskytovat instance, jejichž některá hodnota se dramaticky liší od hodnot atributu ostatních instancí. V takovém případě hovoříme o odlehlém pozorování. Řešení může být buď instanci odstranit, což je ale rozhodnutí, které by mělo být vždy pečlivě zváženo, případně je možné opět hodnotu atributu pro danou instanci nahradit mediánem nebo průměrem hodnot atributu [5].

4.3.6 Škálování vlastností

Další důležitou částí předzpracování dat je škálování vlastností datové sady. Některé algoritmy neumějí dobře zvládat numerické atributy, které mají příliš rozdílné rozsahy hodnot. Pokud například jeden z atributů má hodnoty v rozsahu 0 až 1 a druhý 1 až 1 000 000, pak se algoritmus naučí jen velmi obtížně řešit definovanou úlohu. Jedním z řešení je použití tzv. min-max škálování, někdy též nazývaném normalizací, při kterém dochází k posunutí a škálování hodnot do rozsahu od 0 do 1 [5]. To se provede následujícím vzorečkem:

$$X_{sc} = \frac{X - X_{\min}}{X_{\max} - X_{\min}} \tag{5}$$

Druhým způsobem škálování je tzv. standardizace, která oproti předchozímu typu škálování, neškáluje hodnoty do předem specifikovaného rozmezí hodnot, což může být problém pro některé

algoritmy, jako například neuronové sítě, které často očekávají vstupní hodnoty v rozmezí od 0 do 1. Standardizace je provedena dle následujícího předpisu [6]:

$$\tilde{x}_{ij} = \frac{x_{ij}}{\sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(x_{ij} - \bar{x}_j\right)^2}} \tag{6}$$

kde:

x_{ij} hodnota j-tého atributu i-té instance

$\bar{x}_j$ je průměrná hodnota j-tého atributu datové sady

$\tilde{x}_{ij}$ je standardizovaná hodnota j-tého atributu i-té instance

m je počet instancí datové sady

4.3.7 Výběr modelu

Pokud byly všechny předchozí kroky provedeny správně, nic nám nebrání v samotném výběru, natrénovaní a ohodnocení modelu ML. Obecně se dá říci, že vhodný způsob je začínat na méně složitých, snadno interpretovatelných a rychle naučitelných modelech. Výběr modelu je iterativní proces, při kterém vybereme model, pokusíme se jej naučit na trénovací datové sadě, poté provádíme křížené ověření pro otestování fungování a ladíme model, dokud nejsme s výsledkem spokojeni anebo dokud nepřejdeme na složitější model, od kterého očekáváme lepší výsledky. Poté co nalezneme model, který splňuje naše očekávání, model otestujeme na testovací datové sadě, prezentujeme výsledky a do budoucna ladíme a monitorujeme jeho užívání [5]. V následujících kapitolách se podíváme na některé často používané modely ML a také na samotné učení těchto modelů.

4.4 Učení modelů

Jak již bylo dříve zmíněno, hlavním znakem ML je to, že se jeho modelu dokáží učit. V kontextu ML je učením myšlen proces upravování hodnot parametrů modelu, což má za následek vytvoření modelu, který odpovídá instancím trénovací datové sady. Existují dva obecné přístupy k učení:

- Přímé vypočítání hodnot parametrů modelu z rovnice modelu a hodnot atributů instancí v trénovací datové sadě.
- Iterativní způsob optimalizace řešení, při kterém dochází k postupné úpravě hodnot parametrů modelu ve snaze minimalizovat hodnotu chybové funkce přes celou trénovací datovou sadu.

Pro účely vysvětlení principu učení si vždy nejprve definujeme model ML, na kterém učení bude probíhat.

4.4.1 Lineární regrese

Lineární regrese je nejzákladnější model ML. Jedná se o lineární funkci, jejímž vstupem je jedna nebo více numerických hodnot a výstupem jedna numerická hodnota. Model má dva parametry: θ_1 a θ_2. Slovy vyjádřeno, lineární regrese je model ML, který provádí predikci pomocí vypočítání vážené sumy vstupů a následným připočtením konstanty zvané bias. Matematicky vyjádřeno následující rovnicí [5]:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n \tag{7}$$

kde:

- $\hat{y}$ je predikovaná hodnota
- n je počet atributů
- x_i je i-tá hodnota atributu x
- θ_j je j-tý parametr modelu (zahrnuje bias hodnotu θ_0 a váhy atributů θ_1, θ_2,$\cdots$, θ_n)

V ML se používá spíše vektorový zápis funkce:

$$\hat{y} = h_\theta(x) = \theta^T \cdot x \tag{8}$$

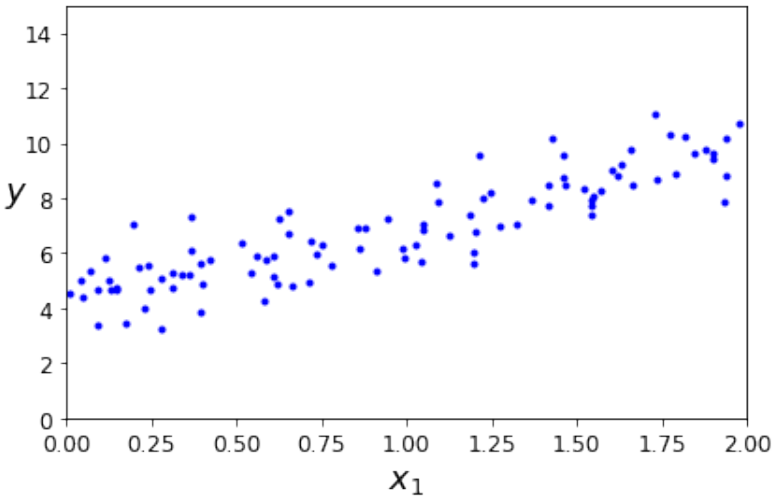
kde:

- θ je vektor parametrů modelu, obsahující hodnotu bias θ_0 a váhy atributů θ_1 až θ_n
- θ^{T} je transponovaný vektor vektoru θ
- x je vektor atributů jedné instance obsahující hodnoty x_0 až x_n, přičemž hodnota x_0 je vždy rovna číslu 1
- $\theta^{\mathrm{T}} \cdot x$ je kartézský součin vektorů θ^{T} a x
- h_θ je funkce hypotézy využívající parametry θ

Při učení je vždy potřeba umět spočítat chybu modelu, pomocí které jsme schopni učení posouvat správným směrem. V případě lineární regrese použijeme funkci MSE (z anglického Mean Square Error), která má následující podobu:

$$\mathrm{MSE}\left(\mathbf{X}, h_\theta\right) = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)^2 \tag{9}$$

Nyní se již můžeme podívat na první způsob učení, tedy přímému vyjádření hodnot parametrů modelu z rovnice modelu.



Obrázek 13: Náhodně vygenerovaná lineární datová sada

4.4.2 Normální rovnice

Pro nalezení hodnoty parametru θ, která minimalizuje hodnotu chybové funkce, lze provést následující výpočet [5]:

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y} \tag{10}$$

kde:

$\hat{\theta}$ je hodnota atributu θ, která minimalizuje hodnotu chybové funkce

y je vektor výsledných hodnot $\mathbf{y}^{(1)}$ až $\mathbf{y}^{(m)}$

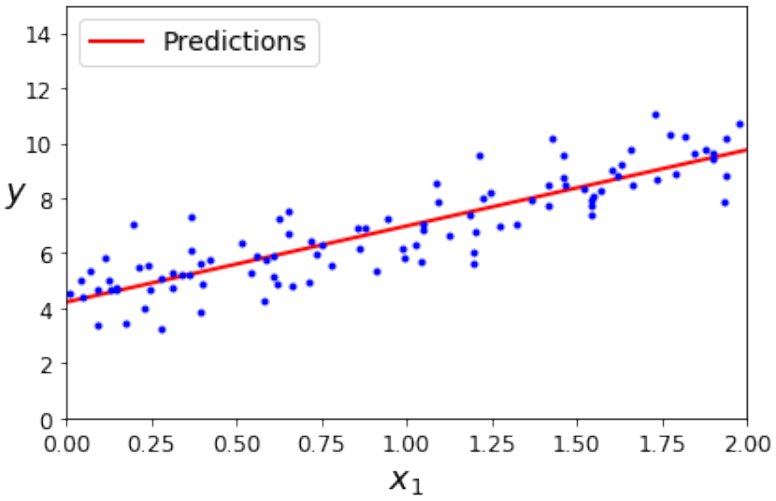
Nyní si na příkladu ukážeme, jak spočítat parametr θ, pokud máme následující zadání:

X – množina 100 číselných hodnot v rozsahu od 0 do 2,

$y = 4 + 3x_1 + \text{random}(0,1)$.

Jak je ze vzorce pro výpočet hodnoty y patrné, k výsledné hodnotě je připočten šum, který nám slouží pro větší variabilitu a realističnost výsledné datové sady. Výsledná datová sada je znázorněna na obrázku 13.

Nyní bychom chtěli vypočítat parametr $\hat{\theta}$, což provedeme dosazením hodnot z naší datové sady do normální rovnice. Výslednými koeficienty jsou $\theta_0 = 3.865$ a $\theta_1 = 3.139$, což je velmi blízko skutečným koeficientům, které jsou $\theta_0 = 4$ a $\theta_1 = 3$. Na obrázku 14 je znázorněna predikce při použití našeho modelu s vypočtenými parametry.



Obrázek 14: Ukázka lineární regrese

4.4.3 Gradient Descent

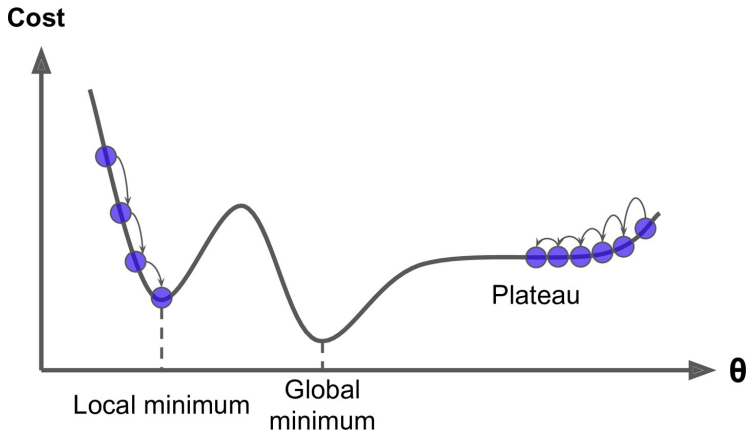
Ne u všech modelů je možné parametry dopočítat, ale musí být nalezeny pomocí optimalizačních algoritmů. Úkolem optimalizačních algoritmů je vyhledávat globální optimum, tedy buď minimum nebo maximum, z oboru hodnot předložené funkce. Pochopitelně někoho může napadnout, vyzkoušet celý definiční obor funkce a posléze vybrat výstup funkce, který je optimální, ale to však ne vždy je možné, jelikož někdy definiční obor funkce neznáme nebo je to časově příliš náročné. Optimalizační algoritmy nám tedy slouží k efektivnímu vyhledání globálního optima funkce [7].

Jeden z velmi často používaných optimalizačních algoritmů je Gradient Descent, který je schopen nalézt optimální řešení pro celou řadu problémů. Hlavní myšlenkou tohoto algoritmu je iterativně posouvat hodnoty parametrů modelu ve snaze minimalizovat chybovou funkci. Jelikož se snaží minimalizovat hodnotu chybové funkce, optimum je v tomto případě minimem.

Algoritmus začíná s náhodně nastavenou hodnotou parametru θ a v každé iteraci cyklu provádí úpravu hodnoty parametru tak, aby docházelo k postupnému snižování hodnoty chybové funkce (například *MSE*), až do té doby, než chyba konverguje k minimu.

Hodnotu, o kterou se parametry posouvají, určuje hyper-parametr koeficient učení. Pokud by byl příliš nízký, velmi dlouho by trvalo, že by algoritmus vyhledal minimum. Naopak pokud by byl příliš vysoký, mohl by při posunutí parametru minimum přeskočit a k jeho vyhledání by nakonec nemuselo dojít [5].

U optimalizačních algoritmů rozlišujeme dva druhy optim: lokální a globální. Lokální optimum je taková hodnota parametru, která při drobné změně vždy zvýší chybu modelu. Globální optimum je hodnota vstupu dané funkce, pro kterou je výstupem funkce nejoptimálnější hodnota z celého oboru hodnot. Rozdíl mezi lokálním a globálním minimem je znázorněn na obrázku 15.



Obrázek 15: Lokální a globální minimum [5]

Existuje více variant tohoto algoritmu a na některé se nyní blíže podíváme.

4.4.4 Batch Gradient Descent

Tento typ algoritmu vyhledává minimum chybové funkce pomocí parciální derivace. Tedy pro každý parametr modelu θ_j počítá, jak se hodnota chybové funkce mění, pokud dojde k malé změně hodnoty parametru θ_j. Parciální derivace nám říká, zda je funkce pro konkrétní vstup rostoucí nebo klesající. Princip algoritmu je tedy zjistit sklon funkce pro danou hodnotu parametru θ_j a upravit tento parametr o malou hodnotu tak, aby nová hodnota chyby byla nižší než ta předchozí. Výpočet parciální derivace chybové funkce [5]:

$$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^{m} \left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)} \tag{11}$$

Případně ve vektorovém zápisu:

$$\nabla_\theta \mathrm{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y}) \tag{12}$$

Je také nutné zmínit, že algoritmus Batch Gradient Descent počítá rovnici 12 nad celou trénovací datovou sadou v každém kroku. Pokud je tedy trénovací datová sada příliš velká, může algoritmus trvat velmi dlouho.

Pokud máme vypočítaný gradientní vektor, jsme schopni vypočítat novou hodnotu parametru modelu:

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta \mathrm{MSE}(\theta) \tag{13}$$

kde:

η je koeficient učení

4.4.5 Stochastic Gradient Descent

Hlavním problémem algoritmu Batch Gradient Descent je fakt, že je potřeba v každém kroku zpracovávat celou trénovací datovou sadu, což může vést k velmi dlouhotrvajícímu učení, pokud bude trénovací datová sada příliš velká. Stochastic Gradient Descent řeší problém tím, že vybírá pouze část datové sady, na které počítá hodnotu chybové funkce, čímž na jednu stranu velmi urychluje výpočet, ale také zvyšuje riziko, že nedojde k nalezení optimální hodnoty parametru modelu, resp. dojde k nalezení dobrého řešení, nikoli optimálního. Pokud by však byla funkce příliš proměnlivá, tak díky vlastnosti algoritmu, že často přeskakuje mezi vzdálenými hodnotami oboru hodnot chybové funkce, může být naopak tento algoritmus méně náchylný k uváznutí v lokálním minimu než Batch Gradient Descent [5].

4.4.6 Polynomiální regrese

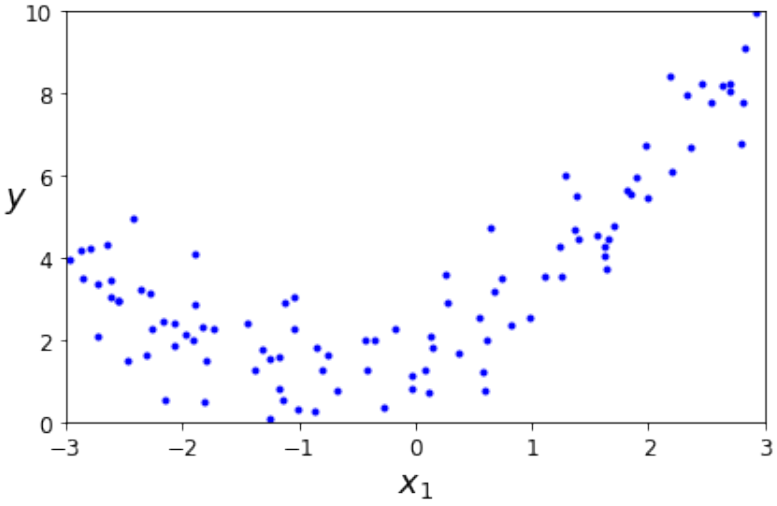
V rámci lineární regrese jsme pracovali s lineárními daty, jejímž modelem byla obyčejná přímka. Pokud by data byla o něco komplexnější a základní lineární model by již nebyl dostačující pro vytváření predikcí, překvapivě by stále bylo možné použít lineární model pro natrénování a následné řešení úlohy. Toho se dá docílit tím, že číselné vlastnosti datové sady se umocní, čímž vznikne nová vlastnost, pomocí které už bude možné provádět predikce [5].

Vezměme si například datovou sadu o 100 instancích a jako funkci použijeme kvadratickou rovnici: $y = 0.5 \cdot x_1^2 + x_1 + 2 + random(0, 1)$. Výsledná datová sada je znázorněna na obrázku 16. Po vizualizaci je jasně vidět, že data nejsou lineární, tedy nebude možné použít klasickou lineární regresi pro vytváření predikcí. Řešením je přidat k hodnotám jejich druhou mocninu, čímž při vykreslení predikcí vznikne namísto přímky parabola, což je znázorněno na obrázku 17. Model odhaduje funkci: $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$, což je poměrně blízké původní kvadratické rovnici.

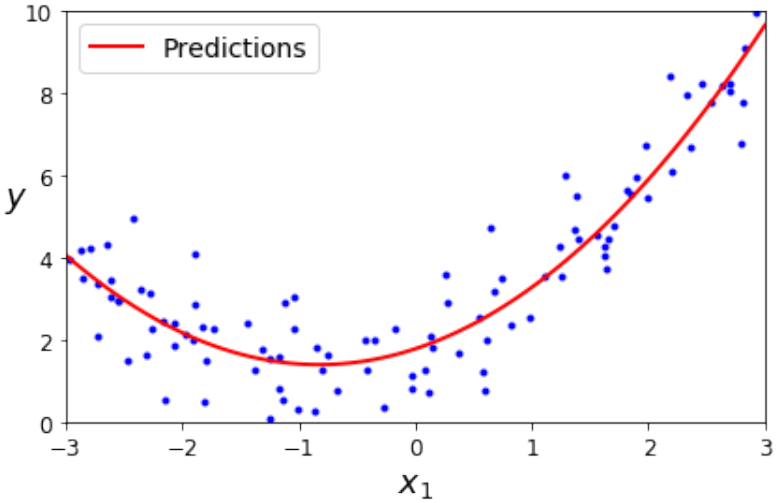
4.4.7 Logistická regrese

Nyní bychom si ukázali, jak se učí jeden z modelů provádět klasifikaci. Logistická regrese je většinou používána k odhadování, s jakou pravděpodobností patří instance datové sady do jednotlivých tříd. Jestliže je pravděpodobnost větší než 50%, pak se předpokládá, že instance patří do dané třídy a algoritmus přiřadí této třídě označení 1, a pokud je menší, pak přiřadí pro tuto třídu označení 0 [5].

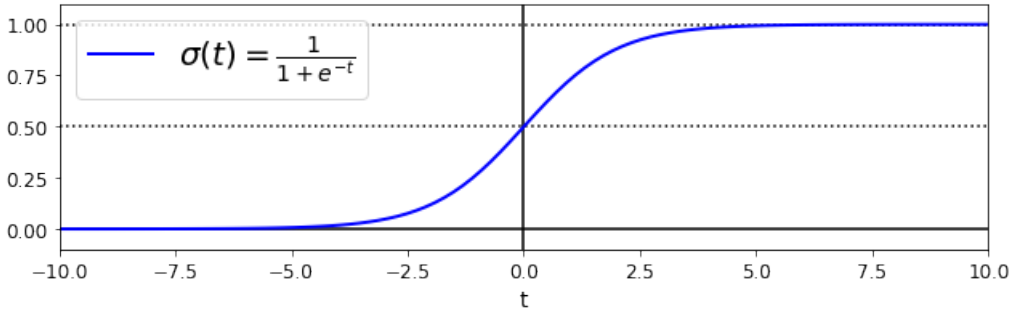
Postup pro odhadování pravděpodobnosti probíhá, stejně jako u lineární regrese, pomocí vážené sumy hodnot vstupních atributů a poté přičtením konstanty. Rozdíl je však ten, že



Obrázek 16: Náhodně vygenerovaná nelineární datová sada



Obrázek 17: Ukázka polynomiální regrese



Obrázek 18: Logistická funkce

namísto okamžitého navrácení výsledku nejprve algoritmus provede tzv. logistický krok. Rovnice pro odhad pravděpodobnosti je následující:

$$\hat{p} = h_\theta(\mathbf{x}) = \sigma\left(\theta^T \cdot \mathbf{x}\right) \tag{14}$$

, přičemž logistický krok je v rovnici značen písmenem σ a jedná se o logistickou funkci neboli sigmoidu. Logistická funkce je dána následujícím předpisem:

$$\sigma(t) = \frac{1}{1 + \exp(-t)} \tag{15}$$

, a je také znázorněna na obrázku 18.

Ve chvíli, kdy má logistická regrese vypočten odhad pravděpodobnosti ve vektorovém tvaru, je možné provádět predikci a to dle následujícího pravidla:

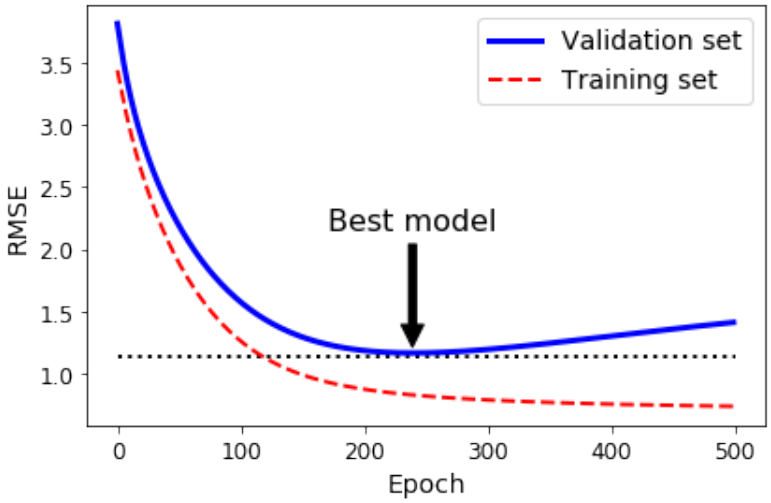
$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases} \tag{16}$$

Dále nám zbývá objasnit, jakým způsobem se model logistické regrese učí. Cílem učení je nastavit parametr θ tak, aby model odhadoval vysoké pravděpodobnosti pro pozitivní instance (kde $y = 1$) a malé pravděpodobnosti pro negativní instance (kde $y = 0$). Chybová funkce pro jednu trénovací instanci je dána následujícím předpisem:

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases} \tag{17}$$

Chybová funkce pro celou trénovací sadu je dána průměrnou chybou všech instancí. Předpis vypadá následovně:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - \hat{p}^{(i)}\right) \right] \tag{18}$$



Obrázek 19: Ukázka včasného zastavení

Pro optimalizaci parametru θ se dá použít algoritmus Gradient Descent, pro nějž potřebujeme znát parciální derivaci chybové funkce logistické regrese. Ta vypadá následovně:

$$\frac{\partial}{\partial\theta_j}\mathrm{J}(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma\left(\theta^T\cdot\mathbf{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)} \tag{19}$$

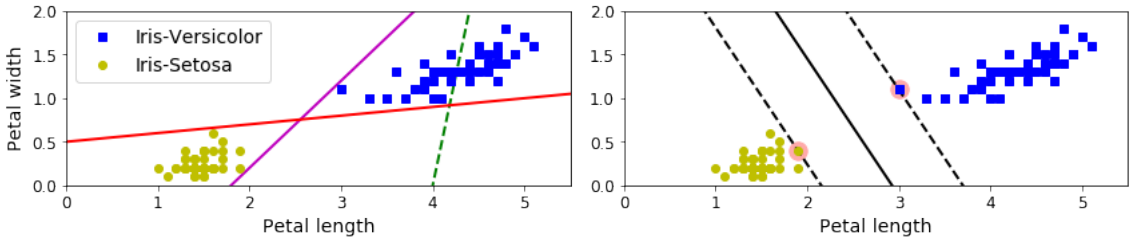
Ve chvíli kdy máme k dispozici všechny parciální derivace, jsme schopni použít algoritmus Batch Gradient Descent, pomocí kterého odladíme model, aby prováděl požadované predikce.

4.4.8 Včasné zastavení

Jedním z problémů souvisejícím s učením modelu ML je problém včasného zastavení. Pokud bychom například používali algoritmus Gradient Descent, chtěli bychom, aby prováděl optimalizaci parametrů modelu tak dlouho, dokud by nenašel minimum chybové funkce, avšak jednalo by se o chybovou funkce aplikovanou na trénovací datové sadě. Typicky při učení modelu dochází k postupnému zmenšování chyby na trénovací datové sadě a současně zmenšování chyby na validační datové sadě, přičemž v určitý okamžik se chyba na validační sadě ustálí a poté se může začít zvyšovat, přičemž chyba na trénovací datové sadě dále klesá. Cílem včasného zastavení je najít optimum na validační datové sadě, což velmi pomáhá generalizaci modelu [5]. Průběh učení a včasné zastavení je znázorněno na obrázku 19.

4.5 Support Vector Machines

SVM je velmi silný model ML, který je schopen provádět klasifikaci a regresi na lineárních i nelineárních datech. Často bývá SVM dobrou volbou pro klasifikaci komplexních problémů s malou nebo středně velkou velikostí datové sady [5].



Obrázek 20: Klasifikace s velkým odstupem

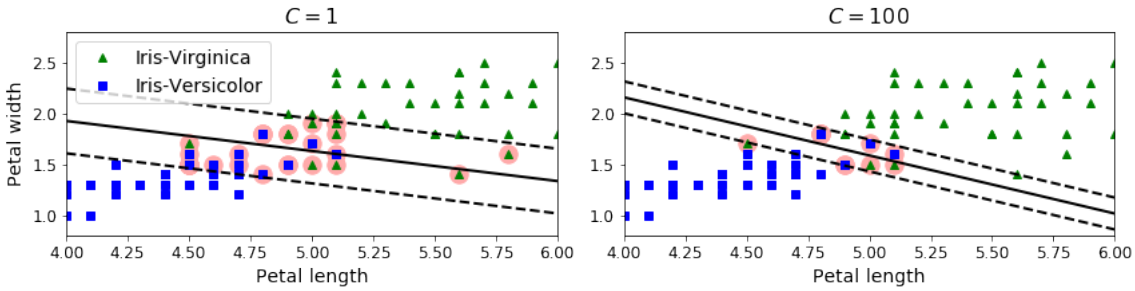
4.5.1 Klasifikace

Na obrázku 20 je vidět ukázka fungování SVM a porovnání s ostatními klasifikátory. Vlevo jsou znázorněny tři různé klasifikátory, přičemž první, znázorněný zelenou čárkovanou čarou, rozděluje třídy naprosto zle, ostatní dva oddělují různými způsoby třídy zcela bezchybně, ale modely jsou vedeny příliš blízko jednotlivých instancí, což může vést k nepřesnostem při použití jiné než trénovací datové sady. Na rozdíl od toho na obrázku vpravo je znázorněna klasifikace pomocí SVM. Nejenže je provedena klasifikace bez chyby, ale ještě k tomu model odděluje třídy s co možná největším odstupem od jednotlivých instancí datové sady.

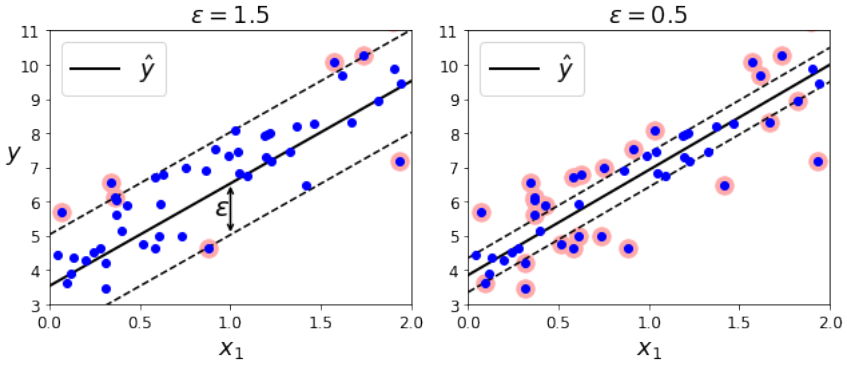
Tento způsob se nazývá klasifikace s velkým odstupem (z anglického large margin classification). Ohraničená mezera mezi třídami se nazývá ulice (z anglického street). Instance ležící na hraně ulice se nazývají podpůrné vektory (z anglického support vectors). Pokud instance datové sady jsou striktně odděleny, tedy žádná instance neleží na ulici, klasifikace se nazývá klasifikace se silným odstupem (z anglického hard margin classification). Samozřejmě ne vždy jsou instance striktně oddělitelné jako v předchozím případě a v takovém případě hovoříme o klasifikaci s částečným odstupem (z anglického solf margin classification). Klasifikace se silným odstupem je možná pouze pokud jsou data lineárně oddělitelná a také pokud neexistují odlehlá pozorování. Pokud tomu tak není, nelze dosáhnout striktního rozdělení tříd a musíme vyvážit potřebu mít co nejširší ulici mezi třídami a současně omezit počet instancí, které leží přímo na ulici. K tomu slouží v SVM hyper-parametr C, který čím je menší, tím širší je výsledná ulice, ale také více instancí na ulici. Rozdíl mezi klasifikací s vysokou a nízkou hodnotou hyper-parametru C je znázorněn na obrázku 21.

4.5.2 Regrese

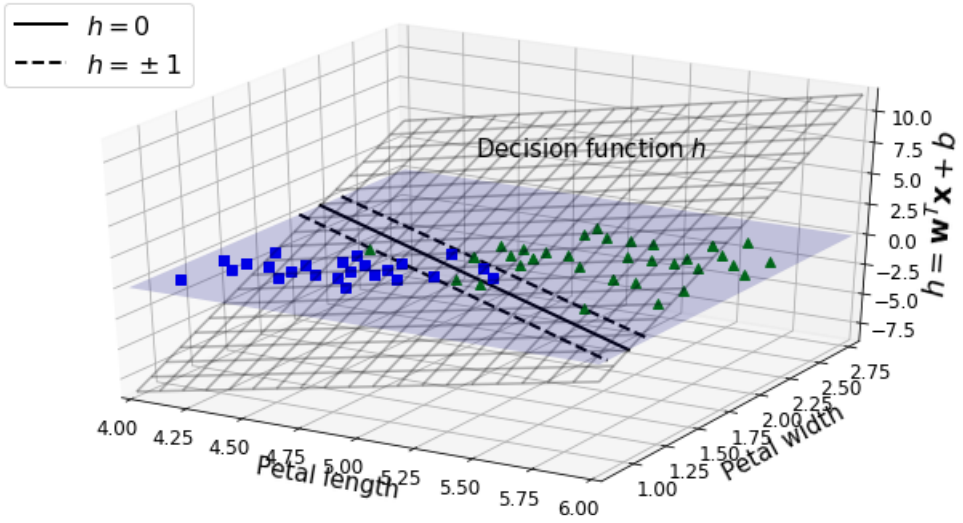
U regrese pomocí SVM není cílem vytvořit co nejširší ulici mezi dvěma třídami s minimalizací počtu instancí ležících na ulici, ale je cílem dostat na ulici co možná nejvíce instancí a minimalizovat počet instancí ležících mimo ulici. Šířka ulice je kontrolovaná hyper-parametrem ε. Příklad regrese je znázorněn na obrázku 22.



Obrázek 21: Rozdíl mezi slabším a silnějším odstupem při klasifikaci pomocí SVM



Obrázek 22: Regrese pomocí SVM



Obrázek 23: Ukázka klasifikace pomocí SVM

4.5.3 Vytváření predikcí

V případě SVM se provádí predikce třídy nové instance x pomocí funkce:

$$\mathbf{w}^T \cdot \mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b \tag{20}$$

kde:

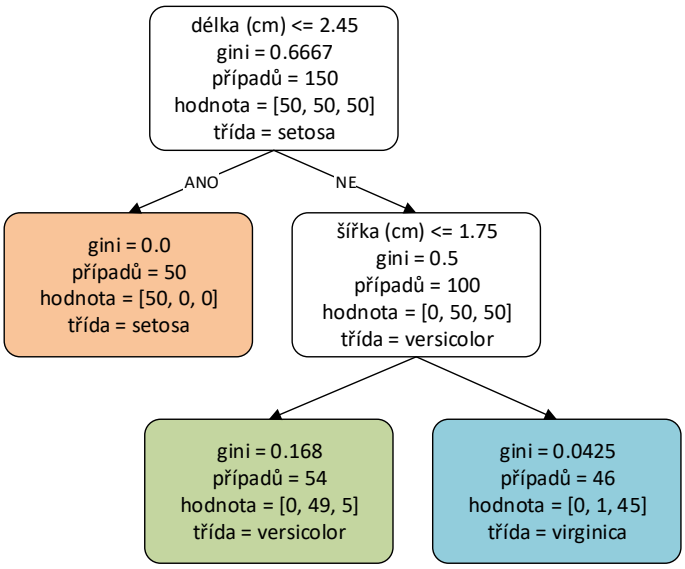
b je bias

w je vektor atributů datové sady

Jestliže je výsledek nezáporný, instance spadá do třídy *1*, jinak do třídy *0*. To se dá znázornit předpisem:

$$\hat{y} = \begin{cases} 0 \text{ if } \mathbf{w}^T \cdot \mathbf{x} + b < 0 \\ 1 \text{ if } \mathbf{w}^T \cdot \mathbf{x} + b \geq 0 \end{cases} \tag{21}$$

Na příkladu na obrázku 23 máme klasifikaci květin pomocí dvou predikátů, šířky a délky okvětních lístků. Svislá osa představuje výsledek rovnice dané předpisem 20. Rozhodovací prahy jsou sada bodů, pro které je rozhodovací funkce rovna 0 (na obrázku je hodnota 0 znázorněna tlustou černou čarou). Čárkované čáry reprezentují body, pro které je výsledek rozhodovací funkce roven 1, resp. -1. Trénovaní SVM představuje nalezení hodnot w a b, pro které vznikne co nejširší ulice s co nejméně (ideálně žádnými) body ležícími na ulici.



Obrázek 24: Ukázka rozhodovacího stromu

4.6 Rozhodovací stromy

Stejně jako SVM je možné i rozhodovací stromy použít pro klasifikaci, regresi a další typy úloh. V této kapitole si popíšeme, jak je možné rozhodovací stromy natrénovat a následně vytvářet predikce. Na obrázku 24 je ukázka klasifikace květin pomocí rozhodovacího stromu.

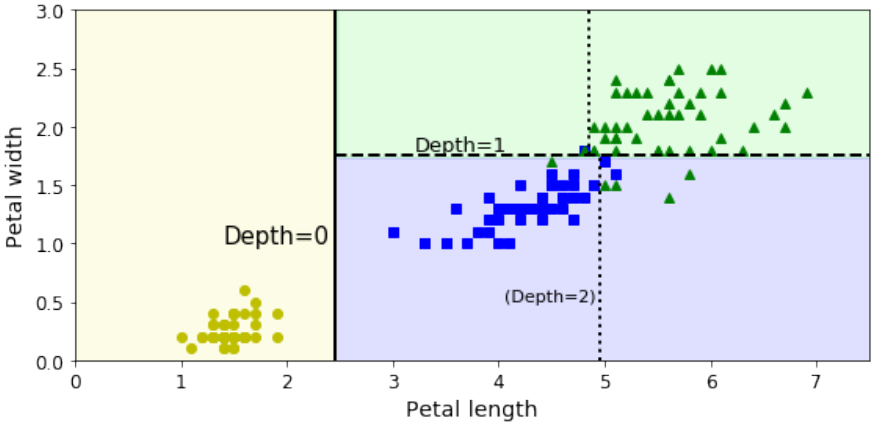
Rozhodovací strom se zpracování od kořene a postupně se dle podmínky přechází to jedné z možných větví. V našem případě je podmínka v kořeni podmínka, že pokud platí, že je délka lístku menší nebo rovna 2.45 cm, pak se pokračuje levou větví a pokud je délka větší, pak se pokračuje pravou větví. Pokud nemá uzel žádného potomka, jedná se tedy o listový uzel, vyhodnotíme, že instance patří do třídy, která je v uzlu uvedena. Pokud má uzel potomky, vždy se na základě podmínky přechází na jednoho z potomků tak dlouho, dokud se nedojde do listového uzlu [5].

Každý uzel obsahuje atribut *případů*, který představuje počet instancí trénovací datové sady, které jsou uzlem zpracovány. V našem případě je například 100 instancí, které mají délku okvětních lístků delší než 2.45 cm. Dále existuje atribut *hodnota*, který udává počet instancí z každé třídy, jež uzlem prochází. Nakonec je v uzlech atribut *gini*, který udává čistotu (z anglického impurity) uzlu. Čistý uzel je takovým uzlem, kterým prochází pouze instance jedné třídy. Výpočet čistoty je prováděn dle následujícího vzorce:

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2 \tag{22}$$

kde:

$p_{i,k}$ je poměr instancí třídy k vůči instancím všech tříd trénovací datové sady v uzlu i



Obrázek 25: Klasifikace pomocí rozhodovacího stromu

4.6.1 Klasifikace

Pokud si například vezmeme uzel v zeleném rámečku z předchozího příkladu, tak u něj bude vypočtena čistota následovně: $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$. Vizualizace klasifikace pomocí rozhodovacího stromu je znázorněna na obrázku 25.

K trénování rozhodovacího stromu se dá použít algoritmus Classification And Regression Tree (CART). Algoritmus nejprve rozdělí trénovací datovou sadu do dvou podmnožin podle jedné vlastnosti k a prahu t_k (například délka okvětního lístku menší nebo rovna 2.45 cm). Vlastnost a práh je určen ve snaze dosáhnout co možná nejčistší podmnožiny. Chybová funkce algoritmu má následující podobu:

$$J\left(k, t_k\right) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}} \tag{23}$$

kde:

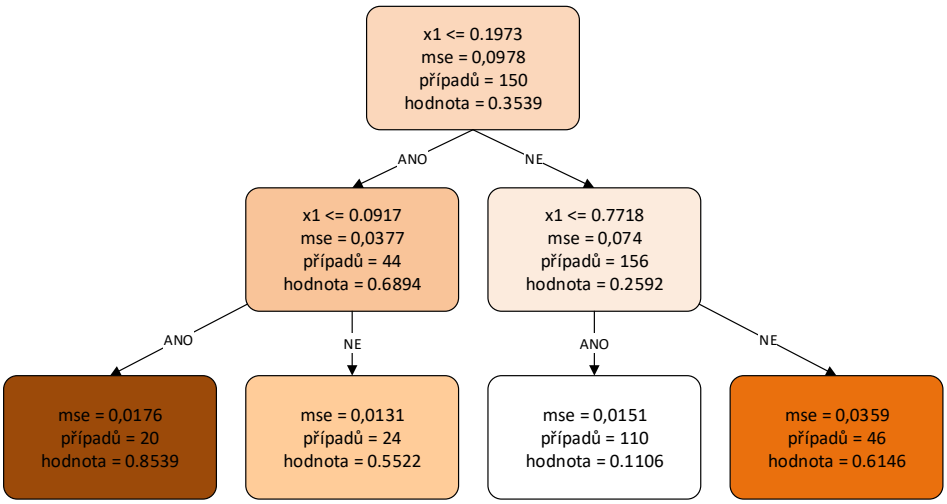
$G_{left/right}$ měří čistotu levé/pravé podmnožiny

$m_{left/right}$ je počet instancí v levé/pravé podmnožině

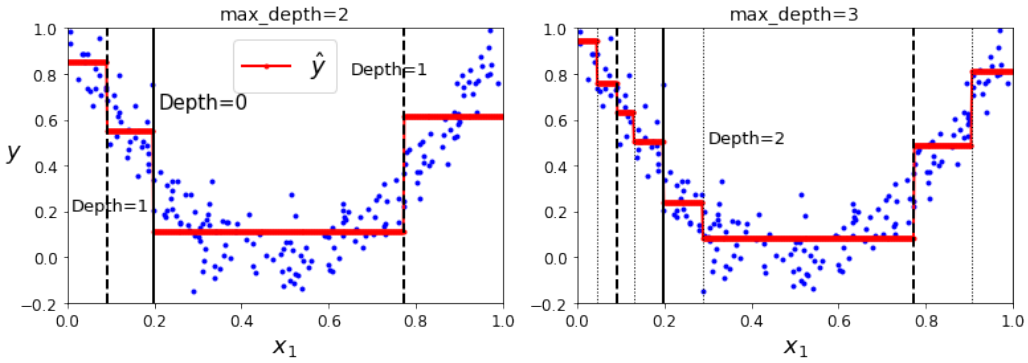
Když dojde k úspěšnému rozdělení trénovací datové sady na dvě podmnožiny, algoritmus tyto podmnožiny rekurzivně opět rozdělí podle stejné logiky jako v předchozím kroku. Rekurze končí ve chvíli, kdy dojde k dosažení maximálního zanoření nebo pokud se žádným dalším rozdělením nezvyšuje čistota.

4.6.2 Regrese

Jak již bylo v úvodu zmíněno, rozhodovací stromy zvládají řešit i regresi. Příklad rozhodovacího stromu řešící problém regrese je vidět na obrázku 26.



Obrázek 26: Ukázka regrese pomocí rozhodovacího stromu

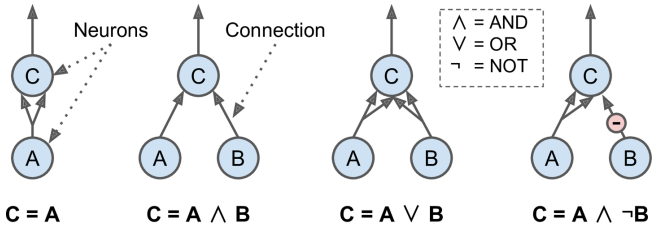


Obrázek 27: Různé zanoření rozhodovacího stromu

Hlavní rozdíl oproti klasifikaci je ten, že místo třídy jsou v uzlech uvedené hodnoty, které predikujeme. Hodnota predikce je v uzlu spočítána jako průměr hodnot všech instancí, které jsou daným uzlem zpracovány. Při učení algoritmus rozděluje jednotlivé regiony trénovacích instancí tak, aby výsledná průměrná hodnota byla co nejpodobnější hodnotám jednotlivých instancí. Na obrázku 27 je znázorněno rozdělení datové sady na regiony, přičemž v levé části je vidět nastavení maximálního zanoření stromu na hodnotu 2 a vpravo na hodnotu 3.

Pro učení je možné opět použít algoritmus CART. Rozdílem je však to, že rozdělování datové sady probíhá za účelem minimalizace MSE. Chybová funkce pro regresi má následující podobu:

$$J\left(k, t_k\right) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}}\ , \text{ kde } \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in node}\left(\hat{y}_{\text{node}} - y^{(i)}\right)^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}}\sum_{i \in node} y^{(i)} \end{cases} \tag{24}$$



Obrázek 28: Neuronové sítě pro provádění logických operací [5]

4.7 Neuronové sítě

Neuronové sítě, přesněji umělé neuronové sítě, jsou inspirované biologickými neuronovými sítěmi, tedy mozkem, i když mnoho expertů se shoduje na tom, že podobnost mezi umělými neuronovými sítěmi a těmi biologickými je skutečně jen v terminologii a možná v pár základních principech fungování. Jedná se o velmi silný, škálovatelný a komplexní nástroj, který je schopen řešit velmi složité problémy ze světa ML, jako například klasifikace miliard kategorií obrázků, rozpoznání mluveného jazyka, doporučování videí na internetu, případně hraní her jako jsou například šachy nebo Go [5].

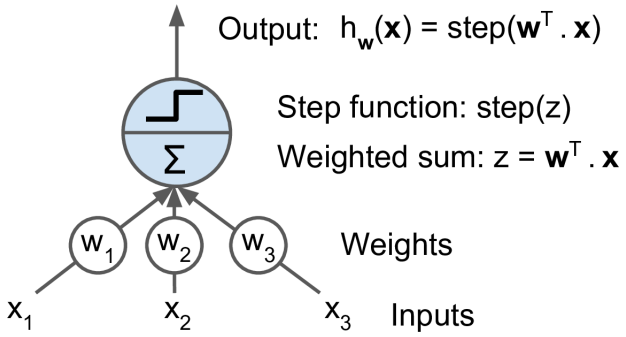
4.7.1 Logické operace pomocí neuronů

Warren McCulloch a Walter Pitts v roce 1943 představili velmi jednoduchý model neuronu, který mohl mít jeden nebo více binárních vstupů a jeden binární výstup. Neuron byl aktivován (nabyl hodnoty 1), kdykoli jeden ze vstupů byl aktivován. Tyto neurony bylo možné propojovat a vytvářet neuronovou síť, pomocí které bylo možné provádět logické operace. Na obrázku 28 je znázorněno několik jednoduchých příkladů neuronových sítí, které provádějí jednoduché logické operace.

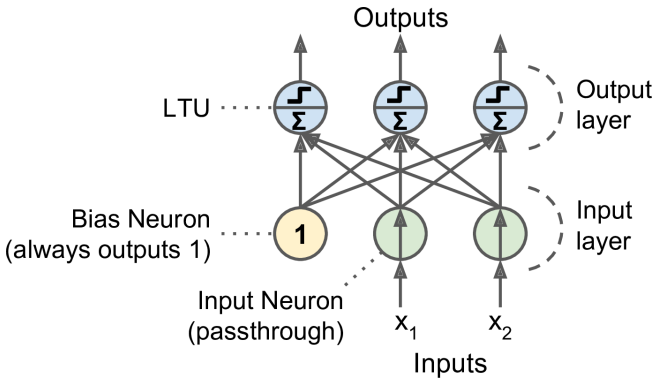
Je zřejmé, že takové neuronové sítě je možné vytvářet i o dost složitější a skládat tak i poměrně komplexní logické výrazy [5].

4.7.2 Perceptron

Perceptron je jednou z nejjednodušších architektur neuronových sítí, která byla uvedena v roce 1957 Frankem Rosenblattem. Koncept neuronu byl o dost rozdílný oproti předchozímu typu. Nyní mohl mít neuron několik číselných, již nikoli binárních, vstupů a několik číselných výstupů. Navíc každý vstup neuronu nebyl neuronem použit přímo, ale byl upraven v závislosti na váze vstupu. Tento typ neuronu se nazývá lineární prahová jednotka (LTU). LTU počítá váženou sumu vstupů, která je vstupem pro tzv. aktivační funkci, jejíž výstup se poté předá na výstup neuronu. Práci LTU znázorňuje obrázek 29.



Obrázek 29: LTU [5]



Obrázek 30: Perceptron [5]

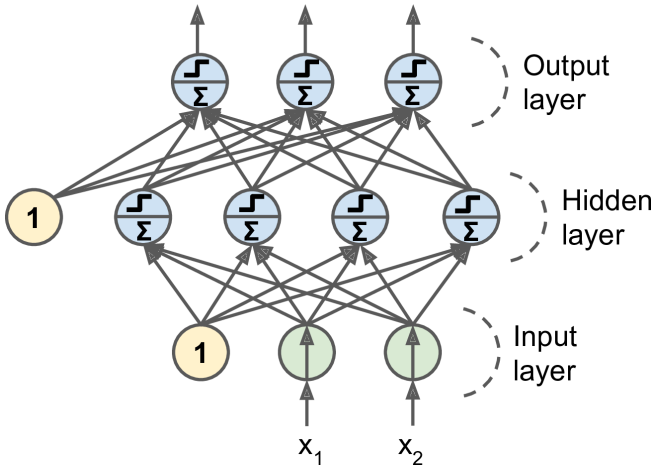
Nejpoužívanější aktivační funkcí perceptronu je tzv. Heaviside step function, která má následující podobu:

$$\text{heaviside}\ (z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z\ \geq 0 \end{cases} \tag{25}$$

Pokud neuronová síť je složena pouze z jedné LTU, pak síť počítá lineární kombinaci vstupů a pokud výsledek přesáhne určený práh, na výstupu získáme kladné číslo, jinak záporné. Učením LTU myslíme vyhledání vhodných hodnot vah LTU.

Perceptron je složen z jedné vrstvy s jednou nebo více LTU, kde každý neuron je propojen se všemi vstupy sítě. Jedna vrstva je vždy přidána na začátek sítě, ve které se nacházejí tzv. vstupní neurony. Navíc je do této vrstvy přidán tzv. bias, což je neuron, jehož výstupem je vždy číslo 1. Perceptron o dvou vstupech a třech výstupech je znázorněn na obrázku 30. Tento perceptron může provádět klasifikaci pro tři různé třídy.

Algoritmus, kterým je možné perceptron naučit, se nazývá Hebbovo učení. Algoritmus vždy vloží do sítě jednu trénovací instanci, provede predikci a pro každý výstup, který vrátil chybnou hodnotu upraví váhu spojení se vstupem tak, aby docházelo ke korekci predikce [5].



Obrázek 31: Vícevrstvá neuronová síť [5]

Funkce pro adaptaci vah má následující podobu:

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta \left(\hat{y}_j - y_j \right) x_i \tag{26}$$

kde:

$w_{i,j}$ je váha spojení mezi i-tým vstupním neuronem a j-tým výstupním neuronem

x_i je i-tá vstupní hodnota aktuální instance trénovací datové sady

$\hat{y}_j$ je výstupní hodnota j-tého výstupního neuronu aktuální instance trénovací datové sady

y_j je cílový výstup j-tého výstupního neuronu pro aktuální instanci trénovací datové sady

η je koeficient učení

4.7.3 Vícevrstvý perceptron a zpětná propagace chyby

Vícevrstvý perceptron je složen z jedné vstupní vrstvy, jedné nebo více vrstev složených z LTU, které se nazývají skryté vrstvy, a jedné výstupní vrstvy. Každá vrstva obsahuje bias a všechny neurony jsou propojeny se všemi neurony následující vrstvy [8]. Ukázka vícevrstvé neuronové sítě je na obrázku 31.

Taková síť je učena pomocí algoritmu zpětné propagace chyby (z anglického backpropagation). Pseudokód algoritmu zpětné propagace chyby je vidět na obrázku 32. Algoritmus nejprve inicializuje váhy vstupů jednotlivých neuronů a začne procházet instance trénovací datové sady. Pro každou instanci dojde k tzv. dopřednému plnění sítě, tedy získání výstupu z neuronové sítě pro danou instanci trénovací datové sady. Poté je spočítána chyba, tedy rozdíl mezi očekávanou a skutečnou hodnotou výstupu.

V dalším kroku se provede adaptace vah výstupní vrstvy sítě podle následujícího předpisu:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i \times g'\left(inpu\ t\_\ \mathrm{sum}_i\right) \tag{27}$$

kde:

$W_{j,i}$ je váha vstupu neuronu i z neuronu j z předchozí vrstvy

α je koeficient učení

a_j je aktivační hodnota neuronu j, neboli hodnota výstupu neuronu j

Err_i je rozdíl mezi výstupem neuronové sítě a očekávaným výstupem z trénovací datové sady

g' je derivace aktivační funkce

$inpu\ t\_\ \mathrm{sum}_i$ je vážená suma vstupů neuronu i

Adaptace vah proběhne pro všechny neurony výstupní vrstvy. Změna váhy vstupu neuronu i z neuronu j z předchozí vrstvy je moderována vynásobením koeficientem učení.

Pro zjištění váženého vstupu aktivační funkce neuronu i, spočítáme kartézský součin vektoru vstupních vah W_i a vektoru výstupů aktivačních funkcí z předchozí vrstvy $\mathbf{A}_i$ a následně připočteme hodnotu bias, tedy platí následující vztah:

$$input\_sum_i = \mathbf{W}_i \cdot \mathbf{A}_i + b \tag{28}$$

Výstupní hodnotu aktivační funkce neuronu j získáme následovně:

$$a_j = g\left(\mathrm{input}\_s\ u\ m_j\right) \tag{29}$$

Chyba e výstupu sítě pro instanci trénovací datové sady i označíme jako Err_i. Derivaci aktivační funkce označíme jako $g'(x)$. Nyní můžeme spočítat změnu potřebnou pro úpravu vah:

$$\Delta_i = Err_i \times g'\left(inpu\ t\_s\ u\ m_i\right) \tag{30}$$

Následně spočítáme váhy následujícím předpisem:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i \tag{31}$$

Nyní algoritmus začne procházet jednotlivé skryté vrstvy zpětně a upravovat jednotlivé váhy vstupů neuronů. Výpočet chyby pro úpravu vah skrytých vrstev je dán vzorcem:

$$\Delta_j \leftarrow g'\left(\mathrm{input}\_\ \mathrm{sum}_j\right) \Sigma_i W_{j,i} \Delta_i \tag{32}$$

Algoritmus 2: Backpropagation

Input: neuralNetwork, trainingDataSet, learningRate
Output: Neural network with updated weights

```
1  neuralNetwork <- initialize weights (randomly);
2  while neuralNetwork has not converged do
3      foreach trainingInstance in trainingDataSet do
4          currentOutput <- computeOutput( neuralNetwork, trainingInstance );
5          error <- targetOutput - currentOutput;
           // update the weights leading to the output layer
6          W_{j,i} ← W_{j,i} + α × a_j × Err_i × g'(inpu t_− sum_i);
7          foreach layer in neuralNetwork do
               // compute the error at each node
8              Δ_j ← g'(input_sum_j) Σ_i W_{j,i} Δ_i;
               // update the weights leading into the layer
9              W_{k,j} ← W_{k,j} + α × a_k × Δ_j;
10         end
11     end
12 end
13 return neuralNetwork;
```

Obrázek 32: Pseudokód algoritmu pro zpětnou propagaci chyby [8]

Nakonec můžeme aplikovat předchozí výpočty k dosažení adaptace vah vstupů neuronů ve skrytých vrstvách:

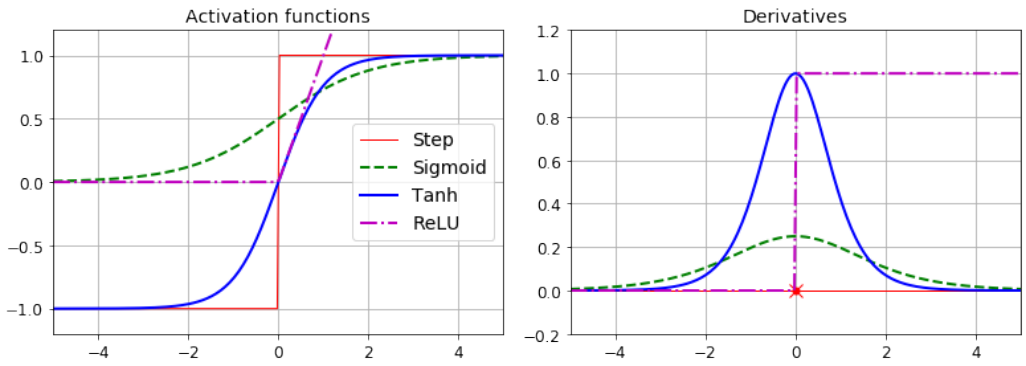
$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \tag{33}$$

Kromě způsobu učení je u vícevrstvého perceptronu změna i v aktivační funkci, ta byla změněna na logistickou funkci a to zejména z toho důvodu, že původní funkce neměla gradient, tedy nebylo by možné použít algoritmus Gradient Descent pro učení sítě. Existují však další populární aktivační funkce, které je možné použít [5]:

- **Hyperbolický tangent**
  - Stejně jako u logistické funkce má tvar písmena S, je spojitá, diferencovatelná, ale její výstup je v rozsahu od -1 do 1, což vede k potřebě provádět v každé vrstvě jistou míru normalizace
  - Funkce má následující podobu:

$$\tanh(z) = 2\sigma(2z) - 1 \tag{34}$$

- **Rectified Linear Unit (ReLU)**



Obrázek 33: Aktivační funkce a jejich derivace [5]

- Funkce je opět spojitá, ale není diferencovatelná v bodě 0, kde se sklon naráz rychle změní, což může negativně působit na učení pomocí algoritmu Gradient Descent.
- Funkce má tento tvar:

$$\mathrm{ReLU}(z) = \max(0, z) \tag{35}$$

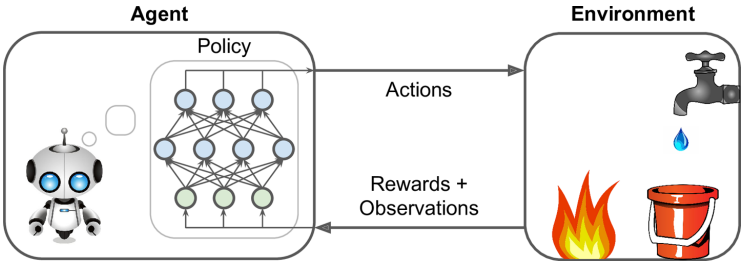
Na obrázku 33 jsou vizualizovány známé aktivační funkce a jejich derivace.

Kromě perceptronu a vícevrstvého perceptronu existuje celá řada dalších architektur neuronových sítí, které obsahují daleko více parametrů a mohou obsahovat různé druhy vrstev. Dohromady tyto neuronové sítě spolu s dalšími modely tvoří podoblast ML zvanou hluboké učení (z anglického Deep Learning). Modely z oblasti hlubokého učení se obecně vyznačují tím, že mají řádově stovky tisíc parametrů, v případě neuronových sítí mohou mít například stovky neuronů propojených stovkami tisíc spojení. Nejznámějšími zástupci těchto neuronových sítí jsou konvoluční neuronové sítě a rekurentní neuronové sítě. Oblast hlubokého učení je však mimo rozsah této diplomové práce, takže se ní dále nebudeme zabývat.

4.8 Posilované učení

Dnes jedním z nejzajímavějších typů ML a současně jedním z nejstarších je posilované učení. Jedním z největších úspěchů posilovaného učení byl v roce 2016 systém AlphaGo, který porazil světového šampiona ve hře Go. Za tímto úspěchem stojí velmi chytré provázání hlubokého učení s posilovaným učení. Mezi nejznámější techniky z posilovaného učení patří policy gradients a deep Q-networks (dále jen DQN) spolu s Markov decission processes (dále jen MDP), a proto se jim budeme také v této kapitole věnovat.

V posilovaném učení existuje softwarový agent, který umí vytvářet pozorování, vybírat mezi možnými akcemi z prostředí a následně vyhodnocovat odměnu z provedených akcí. Cílem je naučit agenta chovat se způsobem, který maximalizuje získané odměny v dlouhodobém horizontu. Odměny mohou být buďto pozitivní anebo negativní, tedy trest [5].



Obrázek 34: Ukázka posilovaného učení s politikou definovanou neuronovou sítí [5]

4.8.1 Hledání politik

Algoritmus používaný softwarovým agentem pro vyhodnocování akcí se nazývá politika. Politikou může být například neuronová síť, která na vstupu získává pozorování a na výstup vrací akci, která má být provedena. Takový příklad je znázorněn na obrázku 34.

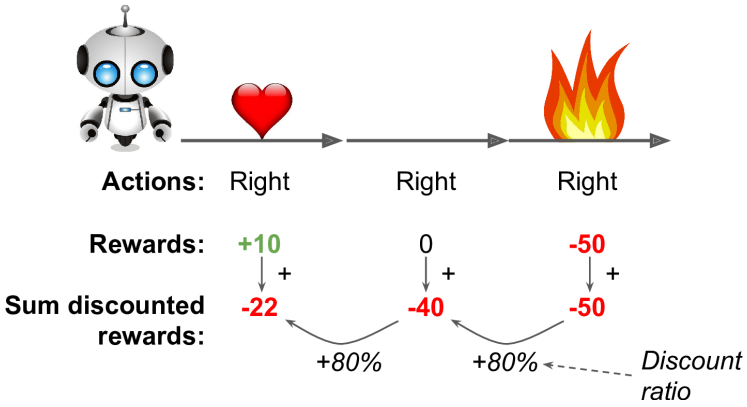
Pokud bychom chtěli agenta naučit pracovat s prostředím za účelem maximalizace odměny, nechali bychom jej prozkoumávat prostředí a provádět akce, přičemž bychom mu vždy po provedení akce předali odměnu, na základě které by se agent mohl rozhodnout, zda příště stejnou akci provede znovu. Tento proces se nazývá hledání politik, v tomto případě konkrétně hledání politik hrubou silou. Pokud by však byl celkový počet možných politik příliš velký, bylo by velmi obtížné dojít k optimálnímu řešení.

Trochu sofistikovanější způsob pro hledání politik je použití genetického algoritmu. Například bychom vytvořili náhodnou generaci 100 různých politik, vyzkoušeli je, ohodnotili jejich kvalitu a poté bychom zničili 80 nejhorších a 20 zbylých bychom drobně upravili a vygenerovali z nich další generaci 100 různých politik. Takto bychom využívali procesu evoluce do té doby, než bychom našli politiku s maximální možnou dosažitelnou odměnou. Dále se dají použít i další způsoby hledání politik, například pomocí postupného upravování jediné politiky ve snaze docílit vyšší odměny, tento způsob se nazývá policy gradients [5].

4.8.2 Problém přidělení odměny

Na rozdíl od učení s učitelem neznáme předem, která akce je vhodná k použití při jednotlivých pozorováních. Jediné, co agent bezpečně zná, je dosažená odměna po provedení nějaké akce. Dalším problémem je však také zpoždění získaných odměn, kdy důsledky provedených akcí nemusejí být patrné okamžitě po jejich provedení, ale mohou se dostavit až později. Agent tedy získá odměnu, případně trest, ale nemusí si být jistý, za kterou akci je odměna udělena. Tento problém se nazývá problém přidělení odměny.

Jedním z možných řešení tohoto problému je ohodnocování akcí pomocí součtu všech odměn získaných po provedení dané akce s postupným snižováním významu odměny. Příklad takového ohodnocení akce je na obrázku 35, kde pokud se agent rozhodne 3x po sobě vykonat pohyb vpravo, získá poprvé odměnu +10, v dalším kroku získá odměnu 0 a nakonec získá odměnu -50.



Obrázek 35: Ukázka ohodnocení akce s postupným snižováním významu odměny [5]

Pro určení dlouhodobého důsledku akce je každá z odměn vynásobena tzv. koeficientem snížení r (z anglického discount rate), který je v našem případě 0.8, a následně vytvořen součet snížených odměn. Pro první akci je tedy celkové skóre vypočítáno následovně: $10+r\times 0+r^2\times(-50) = -22$.

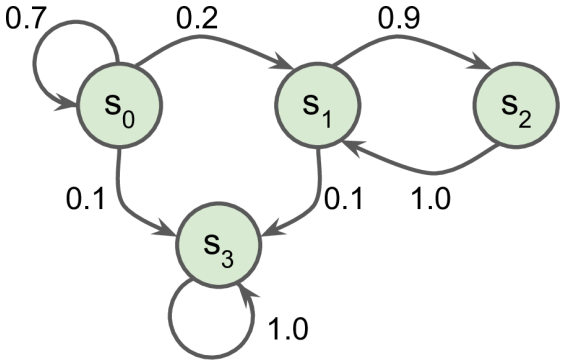
4.8.3 Markov Decision Processes

Již z kraje 20. století vznikl pravděpodobnostní proces bez paměti zvaný Markov chains. Jednalo se o model mající fixní počet stavů, ve kterých je v každém kroku procesu náhodně vybírán přechod na příští stav. Pravděpodobnost přechodu ze stavu s_0 do stavu s_1 je určena fixně, bez ohledu na předchozím navštíveném stavu (systém bez paměti).

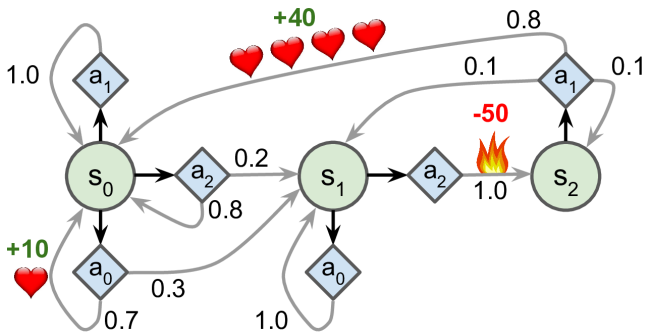
Na obrázku 36 je ukázka systému Markov chains se čtyřmi stavy. Proces začíná ve stavu s_0 se 70% pravděpodobností, že v dalším kroku systém setrvá ve stejném stavu. Pokud by došlo k opuštění stavu, nedalo by se již do něj vrátit, jelikož neexistuje přechod z ostatních stavů do stavu s_0. Pokud by došlo k přechodu do stavu s_1, tak by se v dalším kroku přecházelo do stavu s_2 s 90% pravděpodobností a hned poté se 100% pravděpodobností do stavu s_1. Proces by skončil přechodem do stavu s_3, z nějž není možné přejít do jiného stavu a je tedy koncovým stavem [5].

MDP byl poprvé představen roku 1950 Richardem Bellmanem. Proces je velmi podobný Markov chains s tím rozdílem, že v každém kroku vybírá mezi více možnými akcemi a pravděpodobnosti přechodu závisejí na zvolené akci. Navíc některé přechody mohou vracet agentovi odměnu (pozitivní či negativní) a úkolem agenta je najít politiku, která maximalizuje odměnu v čase.

Příklad MDP je na obrázku 37, kde máme 3 stavy a 3 diskrétní akce. Jestliže začínáme ve stavu s_0, agent může vybírat mezi akcemi a_0, a_1 a a_2. Jestliže vybere akci a_1, zůstává ve stavu s_0 bez jakékoli odměny. Jestliže vybere akci a_0, má 70% pravděpodobnost získání odměny +10 a setrvání ve stavu s_0, ale má také 30% šanci na přechod do stavu s_1. Ve stavu s_1 má agent na výběr mezi akcemi a_0 a a_2. Při výběru akce a_0 setrvá ve stavu s_1 a při výběru akce a_2 přejde



Obrázek 36: Ukázka Markov chains [5]



Obrázek 37: Ukázka MDP [5]

do stavu s_2 a získá trest -50. Ve stavu s_2 musí agent zvolit akci a_1, kterou s pravděpodobností 10% setrvá ve stejném stavu, s pravděpodobností 10% přejde do stavu s_1 a s pravděpodobností 80% přejde do stavu s_0 a získá odměnu +40.

Pro určení strategie agenta se používá tzv. optimální hodnota stavu, kterou má přidělenu každý stav s, je značena $V^*(s)$ a představuje součet všech snížených budoucích odměn, které může průměrně agent očekávat po přechodu na stav s, pokud se chová optimálně. Rovnice má název Bellman Optimality Equation a je dána následujícím předpisem:

$$V^*(s) = \max_a \Sigma_{s'} T\left(s,a,s'\right)\left[R\left(s,a,s'\right) + \gamma \cdot V^*\left(s'\right)\right] \quad \text{for all } s \tag{36}$$

kde:

$T\left(s,a,s'\right)$ je pravděpodobnost přechodu ze stavu s do stavu s' při provedení akce a

$R\left(s,a,s'\right)$ je získaná odměna při přechodu ze stavu s do stavu s' při provedení akce a

γ je koeficient snížení

Algoritmus určený pro odhadování optimálních hodnot stavů se nazývá Value Iteration algorithm. Algoritmus nejprve inicializuje optimální hodnoty stavů na hodnotu 0 a následně je iterativně upravuje. Algoritmus má následující podobu:

$$V_{k+1}(s) \leftarrow \max_a \sum_i T\left(s,a,s'\right)\left[R\left(s,a,s'\right)+\gamma \cdot V_k\left(s'\right)\right] \quad \text{for all } s \tag{37}$$

kde:

$V_k\left(s\right)$ je odhadovaná optimální hodnota stavu s v k-té iteraci algoritmu

Znát optimální hodnoty stavů je důležité pro ohodnocení politiky, ale stále nevíme, co má agent dělat pro maximalizaci odměny. Z tohoto důvodu potřebujeme spočítat tzv. Q-hodnoty. Optimální Q-hodnota stavu a akce, značeno $Q^*(s,a)$, je součet snížených budoucích odměn, které může agent v průměru očekávat po dosažení stavu s a výběru akce a, ale před samotným provedením akce. Algoritmus pro zjištění Q-hodnot se nazývá Q-Value Iteration algorithm a má následující podobu:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T\left(s,a,s'\right)\left[R\left(s,a,s'\right)+\gamma \cdot \max_{a'} Q_k\left(s',a'\right)\right] \quad \text{for all } (s,a) \tag{38}$$

Na začátku algoritmus stejně jako v předchozím případě provádí inicializaci všech Q-hodnot na hodnotu 0 a poté hodnoty upravuje dle předpisu 38. Ve chvíli, kdy jsou známy optimální Q-hodnoty, je možné definovat optimální politiku, značeno $\pi^*(s)$, jako výběr akce a s nejvyšší Q-hodnotou pro každý stav s, tedy platí: $\pi^*(s) = \underset{a}{\text{argmax}} Q^*(s,a)$.

4.8.4 Q-Learning

Již jsme si vysvětlili, jakým způsobem najít optimální politiku ve vztahu k maximalizaci odměn, přičemž výsledná politika je vždy vybrat akci s maximální Q-hodnotou. Často však máme problém s tím, že předem neznáme pravděpodobnosti přechodů mezi stavy při provádění akcí, tedy neznáme $T\left(s,a,s'\right)$. A také neznáme odměny získané při vykonání akcí, tedy neznáme $R\left(s,a,s'\right)$. Abychom zjistili výši odměn, musíme v každém stavu provést každou akci alespoň jedenkrát a pro pravděpodobnosti přechodů musíme provést každou akci v každém stavu opakovaně [5].

Algoritmus Q-Learning slouží pro adaptaci algoritmu Q-Value Iteration na situaci, kdy nejsou pravděpodobnosti přechodů a odměny za akce předem známy. Algoritmus má následující podobu:

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\left(r+\gamma \cdot \max_{a'} Q_k\left(s',a'\right)\right) \tag{39}$$

Pro každý stav a akci algoritmus uchovává průměrnou odměnu získanou při opouštění stavu provedením dané akce současně s odměnou očekávanou v budoucnu. Tento algoritmus patří do kategorie bez politiky, neboť při trénovaní se politika zcela ignoruje a provádějí se akce buď naprosto náhodně anebo jen s částečným přihlédnutím na Q-hodnoty.

Jelikož hledání politiky v MDP pomocí zcela náhodného výběru akcí může trvat dlouho, používá se tzv. *ε-greedy policy*, při kterém dochází k náhodnému výběru akce s pravděpodobností ε a s pravděpodobností 1-ε dojde k výběru akce s nejvyšší Q-hodnotou. Často se začíná s hodnotou ε velmi vysokou, například 1, a postupně se snižuje až na hodnotu velmi nízkou, například 0.05. Výslednou politiku si můžeme představit jako tabulku, kde řádky představují jednotlivé stavy a sloupce představují akce. Q-hodnoty jsou uvedeny v jednotlivých buňkách tabulky.

Problémem algoritmu Q-Learning je ten, že v případě MDP s velkým množstvím stavů a akcí může trvat nalezení optimální politiky velmi dlouho. Možným řešením je použít funkci, která bude Q-hodnoty pouze odhadovat. Tento typ Q-Learningu je nazýván Approximate Q-Learning. Pro odhad Q-hodnot je používána hluboká neuronová síť, která, jelikož slouží pro odhad Q-hodnot, se nazývá DQN. Algoritmus odhadující Q-hodnoty pomocí DQN se nazývá Deep Q-Learning.

Pro učení Deep Q-Learning modelu používáme dvě DQN, kde první slouží pro provádění akcí (nazvěme ji *actor*) a druhá slouží pro pozorování a hodnocení první sítě (nazvěme ji *critic*). Ve stanovených intervalech je vždy *actor* nahrazen sítí *critic*. Pro učení sítě *critic* jsou ukládány veškeré zkušenosti získané sítí *actor* do paměti. Při intervalech nahrazování sítě dojde k načtení dávky zkušeností z paměti a odhadu Q-hodnot, čímž jsme schopni převést tento problém na klasické učení s učitelem, kde síť *critic* se snažíme naučit vytvářet predikce Q-hodnot dle skutečných Q-hodnot získaných z paměti. Pro trénování DQN se používá následující chybová funkce:

$$\begin{gathered} J\left(\theta_{\text{critic}}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - Q\left(s^{(i)}, a^{(i)}, \theta_{\text{critic}}\right)\right)^2 \\ \text{kde: } y^{(i)} = r^{(i)} + \gamma \cdot \max_{a'} Q\left(s'^{(i)}, a', \theta_{\text{actor}}\right) \end{gathered} \tag{40}$$

kde:

$s^{(i)}, a^{(i)}, r^{(i)}, s'^{(i)}$ je stav, akce, odměna a příští stav v i-tém načtení zkušeností z paměti

m je velikost dávky zkušeností z paměti

θ_{critic} jsou parametry sítě *critic*

θ_{actor} jsou parametry sítě *actor*

$Q\left(s^{(i)}, a^{(i)}, \theta_{\text{critic}}\right)$ je predikce Q-hodnoty sítě *critic* v i-tém načtení zkušeností z paměti

$Q\left(s'^{(i)}, a', \theta_{\text{actor}}\right)$ je predikce Q-hodnoty sítě *actor* očekávané v následujícím stavu $s'^{(i)}$ při výběru akce a'

$y^{(i)}$ je cílová Q-hodnota v i-té zkušenosti v paměti

$J\left(\theta_{\text{critic}}\right)$ je chybová funkce používaná k trénování DQN *critic*

4.9 Strojové učení v počítačových hrách

Obecně o učení v počítačových hrách se mluví už dlouhou dobu. Základem učení v počítačových hrách je adaptace hry a NPC na chování hráče, přičemž cílem není pouze získat lepší prožitek ze hry, ale také úprava obtížnosti hry na kvalitě umu hráče. NPC mají schopnost získávat informace ze světa hry, na základě kterých jsou schopni vyhodnocovat důsledky svého chování a podle toho se také rozhodují, jaké akce mají provádět. V praxi se při použití učení ve hrách setkáme s celou řadou problémů, které se nám u konvenčních algoritmů vyhnou. Bohužel definice chování NPC v počítačových hrách bývá dost komplikovaná a při použití konvenčních algoritmů máme z velké části pod kontrolou veškerá rozhodnutí, která NPC provádějí. Existuje celá řada pouček a architektur, které nám pomáhají vytvořit velmi sofistikované chování NPC a současně se neztratit v jeho dílčích částech [3].

Pokud začneme používat nějakou formu učení, okamžitě částečně ztrácíme kontrolu nad tím, proč a jak se NPC rozhodují. Pokud navíc nastane nějaký problém, je o dost horší chybu zreprodukovat tak, aby vývojář bych schopen chybu opravit. AI v počítačových hrách se stává hůře předvídatelnou při použití ML. Vždy je důležité si položit otázku, zda je pro nás důležité použít ML a co tím získáme, jak to naši hru zlepší a zda by nebylo možné stejného efektu docílit jinak. Na druhou stranu AI ve hrách s využitím ML může přinést mnoho zlepšení v celkovém herním zážitku. Hráč může získat pocit, že NPC okolo něj skutečně „přemýšlí“.

5 Praktická část

Následující kapitola se věnuje popisu možností využití ML v počítačových hrách a praktické aplikaci dříve popsané teorie v projektu The Other Kosmos. Na začátku kapitoly je představen projekt The Other Kosmos, následně jsou popsány technologie, který byly při implementaci použity, a nakonec je kapitola věnována problémům z oblasti AI a jejich řešení v projektu The Other Kosmos pomocí ML.

5.1 Projekt The Other Kosmos

Projekt The Other Kosmos byl vyvinut v rámci semestrálních a diplomových prací studentů Fakulty elektrotechniky a informatiky na univerzitě VŠB-TUO pod vedením Ing. Davida Ježka, Ph.D. Jedná se o herně-výukové simulační prostředí, silně inspirované hrou *Minecraft*, které v současnosti obsahuje mnoho prvků simulace, procedurálně generovaný terén, domy, stromy a řeky, dále se po virtuálním světě pohybují NPC a hráč je schopen ve světě sbírat materiál, upravovat terén a stavět budovy. V budoucnu by mohl projekt také sloužit pro výuku, například fyziky.

Jelikož je v systému mnoho procedurálně generovaných prvků, bylo vhodné zvážit také procedurální generování NPC. Takové NPC by dodávalo světu další míru různorodosti a zapadalo by do konceptu projektu. NPC, ty existující i nově procedurálně generované, by v takto různorodém světě mohly mít poměrně velký prostor pro využití komplexní umělé inteligence, která by měla být vzhledem k simulačnímu prostředí co možná nejvíce realistická a adaptovatelná na okolní jevy. NPC se pohybují po mapě náhodně, ale pokud se dostanou do blízkosti hráče, začnou na něj útočit, ale příliš neřeší, co z daného souboje pro ně vyplývá, jaký je účel jejich chování. Existuje zde tedy mnoho vodítek, kde všude a za jakým účelem by bylo vhodné do systému implementovat umělou inteligenci pro zajištění sofistikovaného chování NPC.

5.2 Použité technologie

- **Java** – celý projekt The Other Kosmos je vyvinut v programovacím jazyce Java, takže rozšíření systému bylo taktéž implementováno v jazyce Java.
- **LibGDX** – projekt je vyvinut prostřednictvím rámce LibGDX. Multiplatformní vývojový rámec LibGDX je primárně určen pro vytváření počítačových her v programovacím jazyce Java. Kromě jazyku Java rámec využívá i programovacího jazyka C pro některé výkonnostně kritické úlohy a dosažení multiplatformního využití. Je důležité zmínit, že rámec LibGDX je skutečně pouze vývojový rámec, nejedná se o herní engine, který většinou obsahuje i různé druhy nástrojů, editorů a má striktně předdefinovaný způsob vývoje. Na rozdíl od toho rámec LibGDX umožňuje skutečně velmi volný způsob vývoje, kde je možné rámec využít různými způsoby. Rámec nabízí mnoho funkcionality z oblasti grafiky, zvuku,

zpracování uživatelských vstupů, práce se soubory, dále jsou v něm integrovány knihovny pro podporu fyziky, matematických výpočtů a spousta dalších komponent [9].

- **SQLite** – v rámci implementace bylo také potřeba trvale uchovávat data, k čemuž byl nakonec použit relační databázový systém SQLite. SQLite je knihovna vyvinuta v jazyce C, která nabízí funkčnost plnohodnotného relačního databázového systému [10].

- **jOOQ** – pro překlad a spouštění SQL dotazů z objektově orientovaného prostředí bylo použito ORM jOOQ. jOOQ je knihovna určena pro generování zdrojového kódu v programovacím jazyce Java ze schémat SQL databází a pro vytváření a spouštění SQL dotazů s typovou kontrolu přes implementované rozhraní [11].

- **Weka** – pro použití ML v projektu nebyly implementovány modely ML od základu, ale byly použity existující knihovny, které byly integrovány do systému a následně vhodně použity. Weka je balík modelů ML vyvinutý v programovacím jazyce Java, nabízející spoustu matematických funkcí a algoritmů ML [12].

- **DeepLearning4J** – jedná se o knihovnu obsahující zejména modely hlubokého učení a je její součástí také podpora pro posilované učení, k čemuž byla tato knihovna v projektu využita [13].

5.3 Aplikace ML v projektu The Other Kosmos

5.3.1 Učení chůze procedurálně generovaných NPC

Jak již bylo dříve poznamenáno, projekt The Other Kosmos obsahuje procedurálně generovaný terén, strom, řeky a budovy. Vhodným rozšířením projektu by bylo vytvoření procedurálně generovaných postav, které by se posléze pomocí posilovaného učení naučily chodit po světě. Toto učení dává smysl hlavně z toho důvodu, že pokud by byly NPC generované, ale chůze by u nich fungovala zcela shodně, ubíralo by to celkovému dojmu simulace světa. Vzhledem k tomu, že učení chůze je velmi složitý problém k jehož řešení je potřeba nejprve vyřešit mnoho dalších problémů, například definovat jakou sílu může NPC vyvinout jakou částí těla, jak ovlivní pohyb jedné části těla ty ostatní, jak jsou spojeny části těla NPC a co přesně znamená pohyb jedné z těchto částí. Pochopitelně samotné chůzi předchází stání, které také není jednoduché. V systému existuje fyzikální model, ve kterém je možné definovat gravitační sílu působící na jednotlivé předměty světa, včetně částí těl NPC. Na tyto části působí gravitační síla a NPC musí tuto sílu překonat systematickým působením síly na jednotlivé části těla tak, aby těla setrvalo ve vzpřímené poloze. Skutečný systém chůze NPC by byl na samostatnou diplomovou práci, a proto je potřeba si jej zjednodušit na úplné minimum s cílem otestovat možnosti využití posilovaného učení pro řešení takového problému v počítačové hře. Zjednodušený model chůze a jeho naučení pomocí posilovaného učení je jedním z pilířů implementace této práce.

5.3.2 RPG a soubojový systém

RPG (z anglického Role-playing games) je jeden z žánrů počítačových her, ve kterém je hráčům umožněno hrát za smyšlené postavy a prožívat s nimi jejich příběh [14]. Tento žánr hry se také vyznačuje tím, že postavy mají obvykle odlišné vlastnosti, na jejichž základě má postava schopnosti provádět různé činnosti. Pokud by byl do projektu The Other Kosmos implementován tento systém, bylo by možné využít ML k odhadování vlastností nepřátelských NPC, na jejichž základě by se dalo vyhodnocovat nebezpečí, které NPC hrozí. Součástí by pochopitelně byl i soubojový systém, kterým se myslí souboj, jehož výsledek je přímo ovlivněn složením vlastností jeho účastníků. V praxi by se NPC pohybovalo po světě a když by došlo do blízkosti jiného NPC, použilo by model ML k odhadu nebezpečí a rozhodlo by se, zda se dát na útěk anebo s nepřátelským NPC bojovat.

5.3.3 Systém činností na základě zkušeností

Posledním příkladem využití ML v projektu The Other Kosmos by bylo adaptivní rozhodování NPC na základě předešlých zkušeností. Rozhodování může být využito pro celou řadu případů, avšak v rámci implementace dojde k rozšíření projektu o systém úkolů, které některé NPC může nabízet jak hráči, tak i ostatním NPC, a na základě plnění těchto úkolů se NPC, které rozdává úkoly, bude rozhodovat, zda má či nemá nabídnout úkol konkrétnímu NPC. Pokud NPC úkoly plní velmi často, jsou mu nabízeny i další úkoly, jinak mu nabízeny nejsou. Zkušenosti nebudou vázány na konkrétní NPC, ale budou rozšířeny dle vlastností NPC, které budou definovány v rámci RPG systému. Je totiž zřejmé, že pokud NPC má velmi dobré vlastnosti, má velkou šanci se ve světě probojovat a následně úkoly splnit.

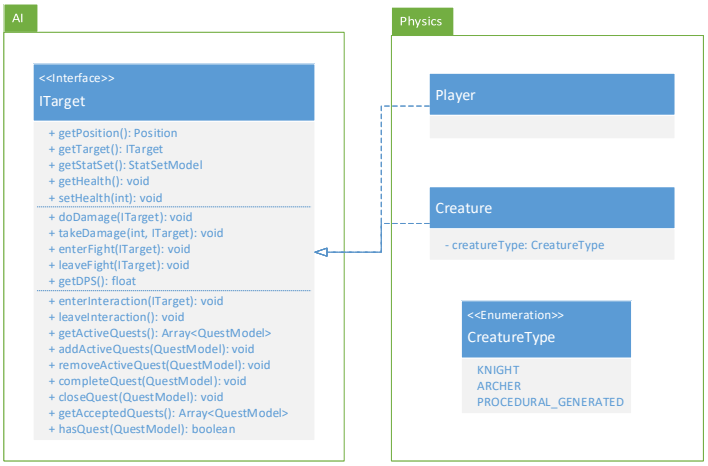
5.4 Návrh a implementace

Všechny úpravy provedené na projektu za účelem vytvoření AI by se daly rozdělit do následujících celků: procedurální generovaní NPC, učení chůze, vytváření NPC ve světě, řízení chování NPC, AI a systém úkolů. Každou část si zde popíšeme.

Aby byla zajištěna nezávislost chování NPC na tom, zda je NPC nuceno reagovat na jiné NPC anebo hráče, vzniklo rozhraní *ITarget*, které implementují třídy *Creature* a *Player*. Instance třídy *Creature* představuje jedno NPC, které se pohybuje po světě, samostatně reaguje na okolní svět a obsahuje vlastní logiku AI. Třída *Player* slouží pro zpracování vstupů hráče, jako je například jeho interakce se světem, odchytávání událostí pro práci s uživatelským rozhraním a další úkony spojené se samotným hraním hry. Rozhraní a implementující třídy popisuje tento diagram 38.

První část metod rozhraní souvisí s obecnými vlastnostmi NPC nebo hráče, další část souvisí se soubojovým systémem a poslední se váže na systém úkolů. NPC může být jeden ze tří typů stvoření:

- *KNIGHT* – jedná se o rytíře, agresivní NPC, které chodí po mapě a útočí na každé NPC, které potká.



Obrázek 38: Rozhraní *ITarget* a jeho implementace

- *ARCHER* – lučištník je neutrální NPC, chodí po mapě a na ostatní NPC nereaguje útokem, ale někteří z lučištníků nabízejí úkoly, které ostatním NPC a hráči nabízejí ke splnění.
- *PROCEDURAL_GENERATED* – jedná se o procedurálně generované NPC, má procedurálně vygenerované tělo a jeho chůze je naučena v samostatné aplikaci pro učení chůze. Chodí po světě a na ostatní NPC reaguje dle zkušeností, nebezpečí a dalších okolností.

5.4.1 Procedurální generování NPC

Nejprve bylo potřeba rozšířit projekt o možnost generování NPC. Generováním NPC se rozumí vytvoření částí těl a následně je poskládat dohromady a uložit toto složení do databáze pro budoucí načtení a použití. Části těla se rozdělují dle typu na hlavu, trup a končetiny. Pro zjednodušení mají všechny nohy stejný tvar i velikost, to stejné platí i o částech trupu. Jednotlivé části trupu jsou spojeny bez možnosti rotace v kloubech, to stejné platí i o jednotlivé části nohou. Jediné klouby těla, které umožňují napojeným částem rotaci, jsou napojení nohou na trup. Generátor těl NPC je popsán zdrojovým kódem 4.

Pro zajištění toho, aby se pro každé NPC nemuselo generovat zvlášť tělo, byly vytvořeny zvířecí druhy NPC, tedy skupiny NPC, které mají společné tělo, název, mají nastavenou velikost populace, mají nastavené podmínky pro vytvoření NPC ve světě a mají vytvořeny RPG vlastnosti. RPG vlastnosti jsou: zdraví, síla, rychlost, výdrž a inteligence. Pro zvířecí druhy NPC vznikl taktéž generátor, který vygeneruje název druhu, RPG vlastnosti a následně použije generátor těla NPC pro vytvoření těla druhu NPC. Zdrojový kód generátoru zvířecích druhů NPC je znázorněn ve zdrojovém kódu 5.

5.4.2 Učení chůze

Po úspěšném vytvoření těla NPC je dalším krokem zajistit naučení pohybu tohoto NPC. Pro tento účel vznikla nová aplikace, ve které je použit velmi podobný svět jako v projektu The Other Kosmos. Po startu této trénovací aplikace dojde k vytvoření NPC dle vygenerovaného předpisu uloženém v databázi a následně k procesu učení chůze. Pro učení chůze byl použit algoritmus posilovaného učení Deep Q-Learning.

Nejprve však bylo potřeba si ujasnit, co přesně bude v našem případě chůzí myšleno. Jak již bylo naznačeno dříve, chůze samotná je dost problematická a nebylo by jednoduché ji implementovat. Nakonec byl zvolen minimalistický model chůze, při které krok končetiny NPC je chápán jako přesunutí z původního místa o kousek blíže k cílové pozici. Není nijakým způsobem řešena potřeba vzpřímené chůze, minimalizace otřesů, poloha hlavy ani další důležité požadavky pro reálnou chůzi. NPC může pohybovat pouze nohami, ostatní části těla jsou ovlivněny pouze gravitační silou, působením jiných částí těla a dalších předmětů.

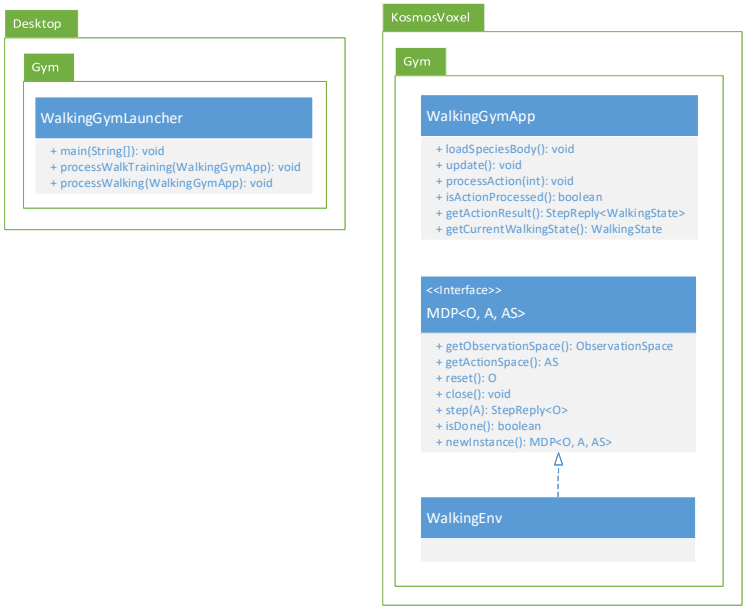
Pro učení chůze je potřeba objasnit způsob určení polohy nohou. Každá noha může být buď v poloze vykročení anebo zakročení ve vztahu k cílové pozici. Z toho vyplývá, že každá noha se může vyskytovat ve 2 stavech, tedy celkový počet stavů je roven číslu 2^n, kde n je počet nohou. Algoritmus pro učení chůze, v našem případě již zmíněný Deep Q-Learning, na vstupu přijímá aktuální stav každé nohy a na výstup vrací žádoucí akci pro provedení. Ty nohy, u kterých má dojít ke změně stavu, budou vykročeny směrem k cílové pozici.

Učení chůze probíhá v samostatné aplikaci, která má název *WalkingGymLauncher*, jejímž úkolem je sestavit LibGDX aplikaci, ve které je následně sestavena plocha a vytvořeno NPC, které se má naučit chodit. Třída obsahuje následující metody:

- *main* – hlavní metoda aplikace, která vytvoří třídu WalkingGymApp a zajistí spuštění LibGDX aplikace.
- *processWalkTraining* – slouží pro spuštění procesu učení chůze a následnému uložení výsledné politiky pro chůzi.
- *processWalking* – slouží pro demonstraci výsledné naučené chůze.

Pro veškerou logiku spojenou s vytvořením světa, vytvořením NPC, zpracováním vstupů a samotným učením chůze slouží třída *WalkingGymApp* (viz diagram na obrázku 39), která obsahuje následující metody:

- *loadSpeciesBody* – načte tělo zvířecího druhu NPC z databáze.
- *update* – metoda pro aktualizace světa, provádí se v každé iteraci hlavního cyklu.
- *processAction* – slouží pro provedení pohybu NPC dle hodnoty parametru (implementace metody ve zdrojovém kódu 3).



Obrázek 39: Třídy obsahující logiku pro učení chůze

- *isActionProcessed* – vrací TRUE, pokud bylo již dokončena akce pohybu vyvolaná metodou processAction.
- *getActionResult* – vrací výsledek provedení kroku chůze, včetně získané odměny a aktuální pozice nohou NPC.
- *getCurrentWalkingState* – vrací aktuální pozici nohou NPC.

Kromě samotného vykonání akce je také důležité konečné vyhodnocení výsledku akce, které má na starost metoda *getActionResult*, jejíž podoba je znázorněna ve zdrojovém kódu 1.

```
public StepReply<WalkingState> getActionResult() {
    Matrix4 bodyTransform = new Matrix4();
    centerBody.getMotionState().getWorldTransform(bodyTransform);
    Vector3 bodyLocation = new Vector3();
    bodyTransform.getTranslation(bodyLocation);
    double reward = previousBodyPosition.dst(movementTarget) - bodyLocation.dst
        (movementTarget);
    return new StepReply<WalkingState>(getCurrentWalkingState(), reward, true,
        null);
}
```

Výpis 1: Metoda pro získání odměny a stavu NPC při učení chůze

Jak je z metody patrné, výsledná odměna je počítána z ušlé vzdálenosti po posledním kroku.

Pro komunikaci mezi trénovacím algoritmem a prostředím světa je použita implementace rozhraní *MDP* nazvaná *WalkingEnv* (třídní diagram na obrázku 39). Třída slouží pro předávání zpráv mezi trénovacím algoritmem a trénovacím prostředím. Třída a rozhraní obsahují následující metody:

- *getObservationSpace* – vrací prohledávací prostor problému, v našem případě vrací všechny možné pozice nohou NPC.
- *getActionSpace* – vrací množinu dostupných akcí, v našem případě všechny kombinace pohybů nohami.
- *reset* – nastavuje proces učení do výchozí pozice.
- *close* – slouží pro uvolnění zdrojů a odpojení od dalších služeb.
- *step* – metoda slouží pro provedení akce zadané parametrem, nejprve dojde k příkazu vykonání akce ve třídě pro učení chůze, následně se vyčká na dokončení procesu kroku a poté funkce vrátí výsledek operace (implementace k dispozici ve zdrojovém kódu 2).
- *isDone* – vrací TRUE, pokud je algoritmus dokončen.
- *newInstance* – vytváří kopii objektu.

```
@Override
public StepReply<WalkingState> step(Integer action) {

   walkingGymApp.processAction(action);

   while (!walkingGymApp.isActionProcessed()) {
      try {
         Thread.sleep(1);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }

    return walkingGymApp.getActionResult();
}
```

Výpis 2: Metoda pro provedení kroku při chůzi NPC a vyčkání na jeho dokončení

5.4.3 Vytváření NPC ve světě

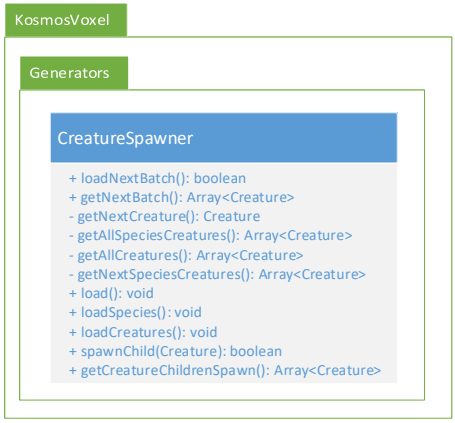
V hlavní aplikaci je hráč vytvořen na výchozí pozici a kolem něj se začne postupně vytvářet svět, ve kterém dochází i k vytváření NPC. Vytváření zvířecích druhů vždy probíhá v určených intervalech a vždy dojde k vytvoření celé populace, tedy všech NPC. Kromě zvířecích druhů je možné vytvářet i NPC jednotlivě, například pro účel vytvoření rozdávačů úkolů, které následně mohou ostatní NPC anebo hráč plnit. Kromě samotného vytváření NPC dochází také k vytváření potomků, přičemž potomci dědí zkušenosti od svých předků. Pro vytváření NPC ve světě slouží třída *CreatureSpawner()* (viz diagram na obrázku 40) s následující metodami:

- *loadNextBatch* – spustí samostatné vlákno, ve kterém dojde k načtení a vytvoření dávky NPC, která obsahuje populaci jednoho zvířecího druhu NPC a jedno samostatné NPC, které splňují podmínky pro vytvoření.
- *getNextBatch* – vrátí vytvořenou dávku NPC, které byly vytvořeny v metodě *loadNextBatch*. Na obrázku 41 je ukázka takto vytvořené skupiny NPC.
- *getNextCreature* – vytvoří jedno NPC z databáze, které splňuje podmínky pro vytvoření.
- *getAllSpeciesCreature* – vytvoření najednou všechny zvířecí druhy NPC z databáze a vrátí je na výstup.
- *getAllCreatures* – vytvoří najednou všechny NPC z databáze a vrátí je na výstup.
- *getNextSpeciesCreatures* – vytvoří populaci pro jeden zvířecí druh NPC v pořadí a vrátí ji na výstup.
- *load* – načte z databáze všechny NPC a zvířecí druhy NPC.
- *loadSpecies* – načte zvířecí druhy NPC z databáze.
- *loadCreatures* – načte všechny NPC z databáze.
- *spawnChild* – vytvoří v samostatném vlákně potomka NPC zadaného parametrem.
- *getCreatureChildrenSpawn* – vrátí potomky NPC vytvořené za dobu od poslední kontroly.

5.4.4 Řízení chování NPC

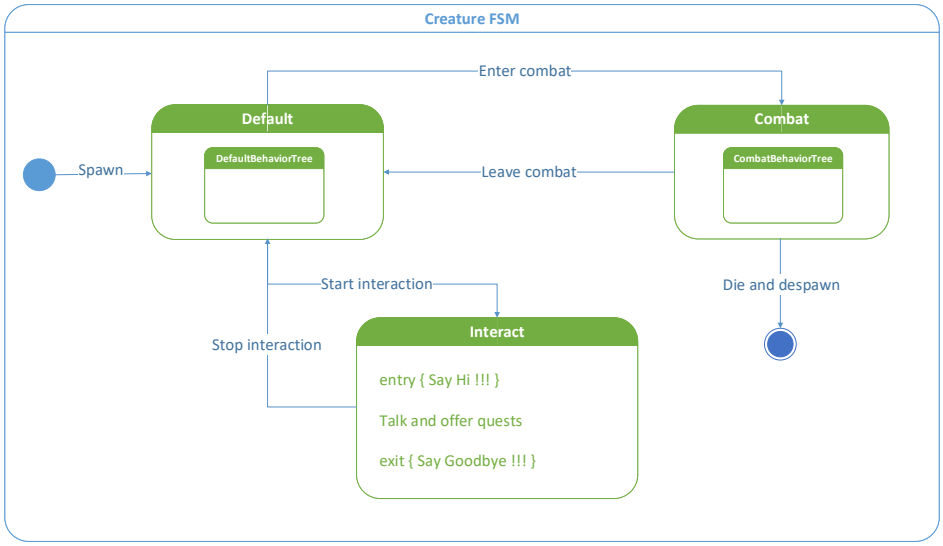
Chování každého NPC je řízeno pomocí FSM znázorněného na obrázku 42. Stavy má automat následující:

- *DEFAULT* – výchozí stav, ve kterém vnitřní logiku určuje behaviorální strom znázorněný na obrázku 43.

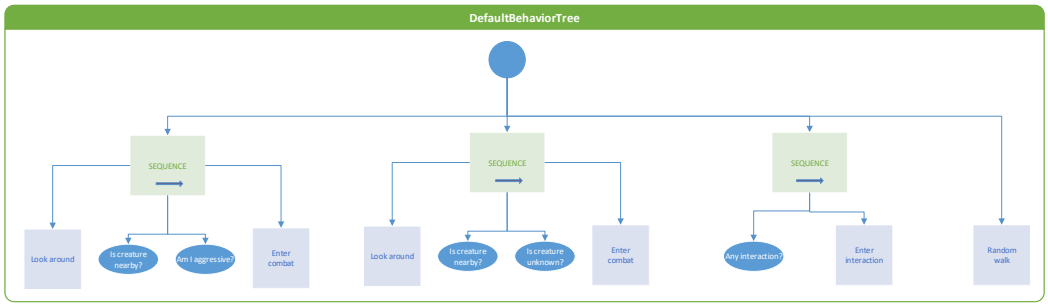


Obrázek 40: Třída pro vytváření NPC ve světě

Obrázek 41: Skupina NPC vytvořená ve světě



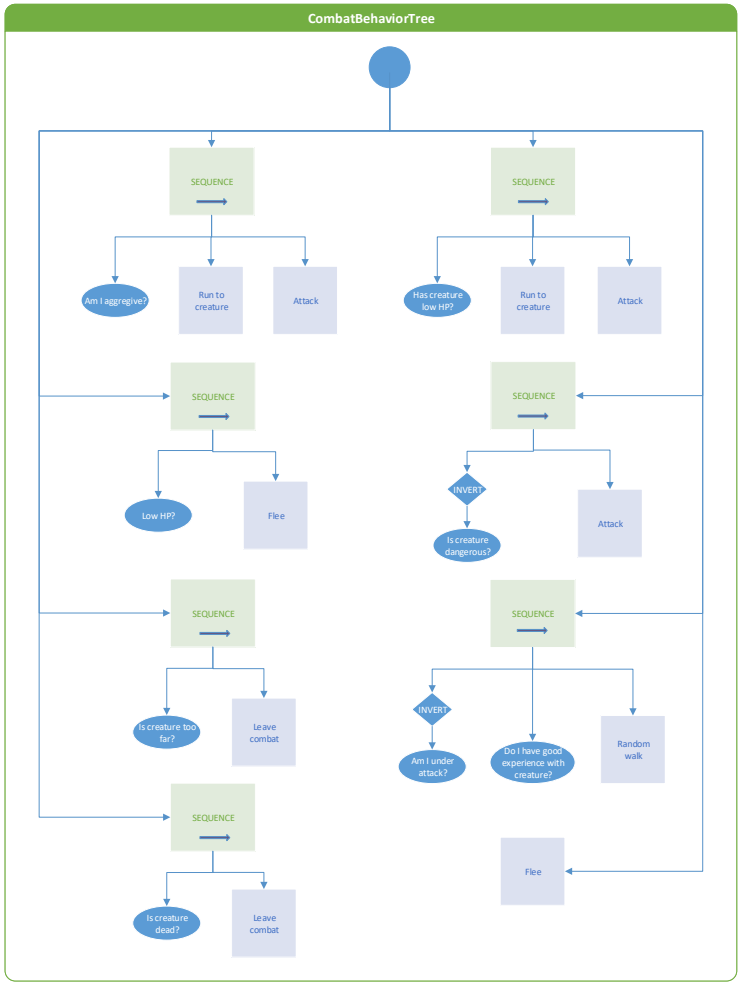
Obrázek 42: FSM pro řízení NPC



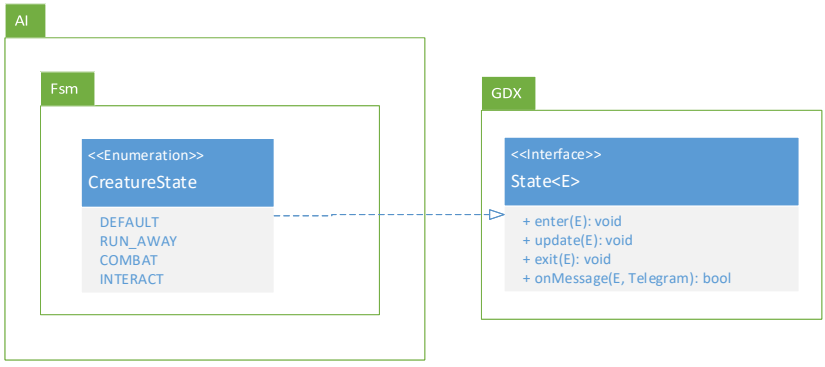
Obrázek 43: Behaviorální strom pro řízení NPC ve výchozím stavu

- *RUN_AWAY* – v tomto stavu NPC utíká od svého nepřítele tak dlouho, dokud od něj není v dostatečně bezpečné vzdálenosti.
- *COMBAT* – stav, ve kterém se NPC nachází pokud je pod útokem anebo samo útočí na jiné NPC nebo hráče, vnitřní logiku určuje behaviorální strom znázorněný na obrázku 44.
- *INTERACT* – stav, ve kterém se NPC nachází, pokud zrovna komunikuje s jiným NPC nebo hráčem. Příkladem může být například rozdávač úkolů, se kterým si hráč promluví a otevře se mu nabídka úkolů.

Pro definici stavů FSM je použito rozhraní z knihovny LibGDX s názvem *State*. Každý stav musí toto rozhraní implementovat a při zpracování FSM je vždy zpracován pouze ten stav, který je právě aktuální. Rozhraní a třída pro stavy NPC jsou zobrazeny diagramem na obrázku 45. Metody rozhraní *State*:



Obrázek 44: Behaviorální strom pro řízení NPC ve stavu souboje



Obrázek 45: Třídní diagram rozhraní a třídy pro implementace stavu FSM

```
#
# Creature default tree
#

# Alias definitions
import lookAround:"com.semestralproject.kosmosvoxel.ai.btree.LookAroundAction"
import
anyCreatureNearby?:"com.semestralproject.kosmosvoxel.ai.btree.AnyCreatureNearbyCond
ition"
import
creatureAggresive?:"com.semestralproject.kosmosvoxel.ai.btree.CreatureAggresiveCond
ition"
import
creatureUnknown?:"com.semestralproject.kosmosvoxel.ai.btree.CreatureUnknownConditio
n"
import enterCombat:"com.semestralproject.kosmosvoxel.ai.btree.EnterCombatAction"
import
anyInteraction?:"com.semestralproject.kosmosvoxel.ai.btree.AnyInteractionCondition"
import
enterInteraction:"com.semestralproject.kosmosvoxel.ai.btree.EnterInteractionAction"
import randomWalk:"com.semestralproject.kosmosvoxel.ai.btree.RandomWalkAction"
import amIAggresive?:"com.semestralproject.kosmosvoxel.ai.btree.AggresiveCondition"

# Tree definition
root
        selector
                sequence
                        lookAround
                        anyCreatureNearby?
                        amIAggresive?
                        enterCombat
                sequence
                        lookAround
                        anyCreatureNearby?
                        creatureUnknown?
                        enterCombat
                sequence
                        anyInteraction?
                        enterInteraction
                randomWalk
```

Obrázek 46: Ukázka definice behaviorálního stromu pro knihovnu LibGDX

- *enter* – slouží pro definici logiky, která se má provést při přechodu na tento stav.
- *update* – volá se při každém zpracování FSM, obsahuje logiku stavu.
- *exit* – slouží pro definici logiky, která se má provést při opuštění tohoto stavu.
- *onMessage* – slouží pro předávání zpráv mezi stavy.

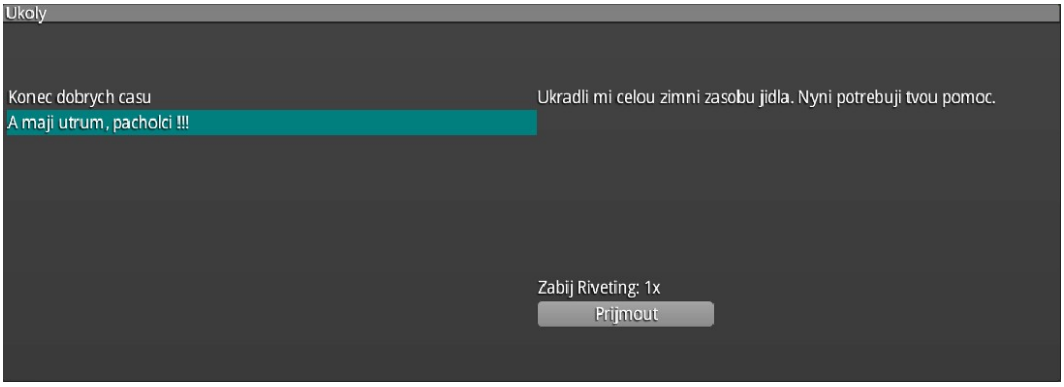
Stejně jako FSM, i behaviorální stromy jsou implementovány v knihovně LibGDX, přičemž jednotlivé větve behaviorálního stromu se zadávají v určeném textovém souboru, ze kterého se poté za běhu aplikace instance behaviorálního stromu vytvoří. Příklad zápisu behaviorálního stromu je znázorněn na obrázku 46.
Behaviorální stromy obsahují následující uzly:

- *CreatureDangerousCondition* – podmínka, zda je nepřátelské NPC nebezpečné pro hráče. Pro vyhodnocení se používá instance třídy FightOrFleeBrainModule registrovaná v mozku NPC.
- *LowHPCondition* – podmínka, zda má NPC množství životů pod kritickou hranicí.
- *FleeAction* – přepne stavový automat chování NPC do stavu RUN_AWAY.

- *CreatureLowHPCondition* – podmínka, zda má nepřátelské NPC méně životů, než je kritická hranice.
- *WalkToTargetAction* – akce pohybu NPC směrem k cíli.
- *AttackAction* – akce provedení útoku na nepřátelské NPC.
- *CreatureDeadCondition* – podmínka, zda je nepřátelské NPC mrtvé.
- *LeaveCombatAction* – přepne stavový automat na stav *DEFAULT*.
- *RandomWalkAction* – akce pro pohyb NPC v náhodném směru.
- *TooFarFromTargetCondition* – podmínka dostatečné vzdálenosti od nepřátelského NPC.
- *AggresiveCondition* – podmínka, zda je samotné NPC agresivní.
- *GoodExperienceWithCreatureCondition* – podmínka, že NPC má s druhým NPC dobrou zkušenost, tedy že na něj nezaútočilo v minulosti příliš často. Pro vyhodnocení podmínky se používá *PredictStartFightingBrainModule* registrovaný v mozku NPC.
- *UnderAttackCondition* – podmínka, zda je NPC pod útokem jiného NPC nebo hráče.
- *LookAroundAction* – NPC si zkontroluje své okolí a vyhledá cíl.
- *AnyCreatureNearbyCondition* – NPC se rozhodne, zda je nepřátelské NPC blízko.
- *CreatureAggresiveCondition* – podmínka, zda je nepřátelské NPC agresivní.
- *CreatureUnknownCondition* – podmínka, která je splněna, pokud NPC a blízké NPC jsou různé typy stvoření a nejedná se o stejný zvířecí druh.
- *EnterCombatAction* – akce pro vstup do souboje s NPC.
- *AnyInteractionCondition* – podmínka, zda se nějaké NPC nebo hráč pokouší o komunikaci s NPC.
- *EnterInteractionAction* – započetí komunikace s NPC nebo hráčem.

5.4.5 Systém úkolů

U NPC je možné definovat, jaké úkoly mohou ostatním NPC a hráči nabízet. Takové NPC je uložené v databázi a má přidělený seznam úkolů, také uložený v databázi. Při vytvořeni takového NPC, NPC ihned nabídne všem NPC své úkoly, přičemž hráč si úkol může vyžádat také, a to sice tak, že přijde do blízkosti NPC a promluví s ním. V takový moment se hráči otevře okno s nabídkou úkolů, ze kterých si může vybrat. Nabídka úkolů je znázorněna na obrázku 47.
Úkoly mohou být třech typů:



Obrázek 47: Okno s nabídkou úkolů

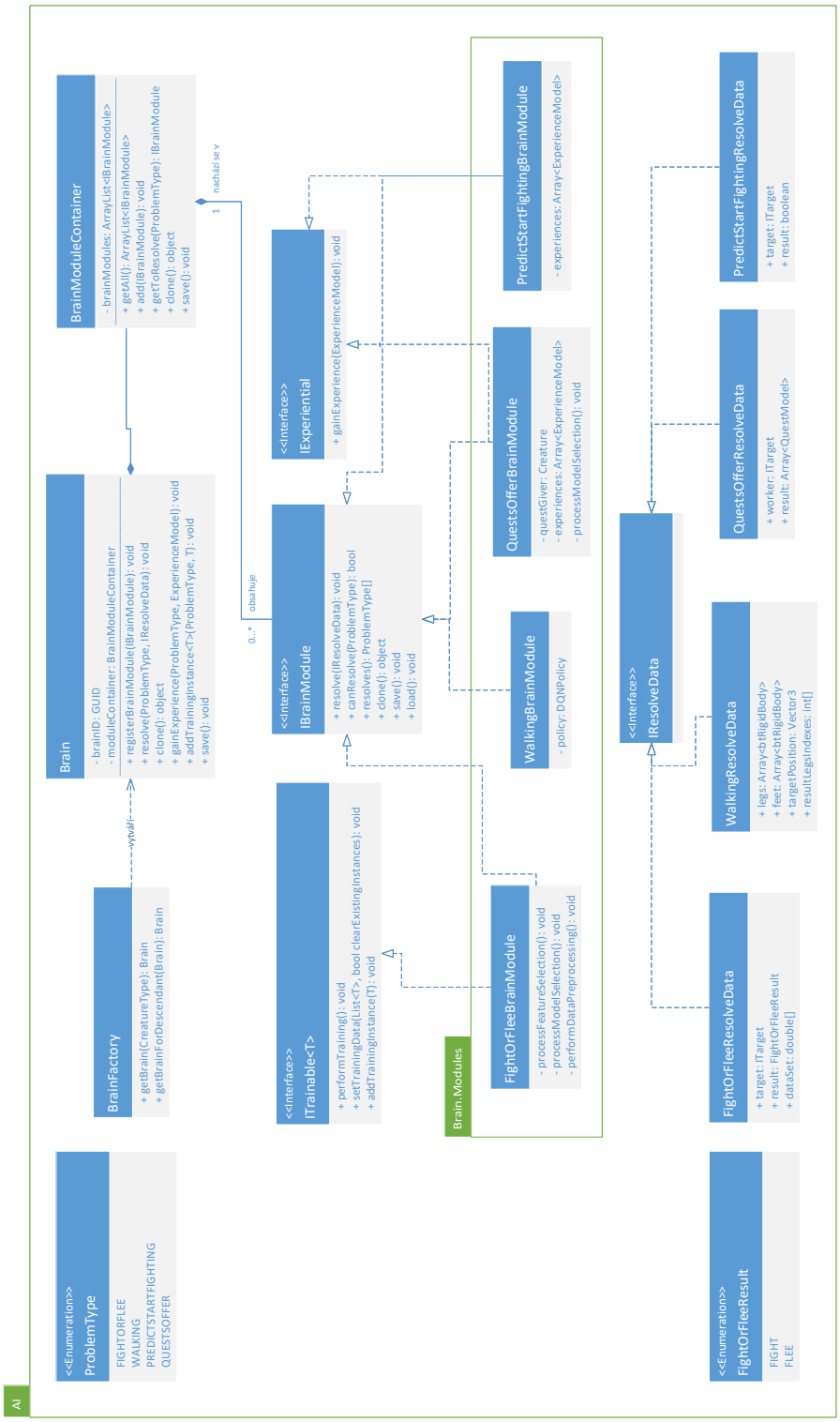
- *KILL* – úkolem je zabít nějaké NPC, typ NPC anebo zvířecí druh NPC.
- *COLLECT* – úkolem je přinést rozdávači úkolů nějaký předmět.
- *INTERACT* – úkolem je promluvit s nějakým NPC, typem NPC anebo zvířecím druhem NPC.
- *EXPLORE* – úkolem je prohledat zadanou oblast.

5.4.6 Modul AI

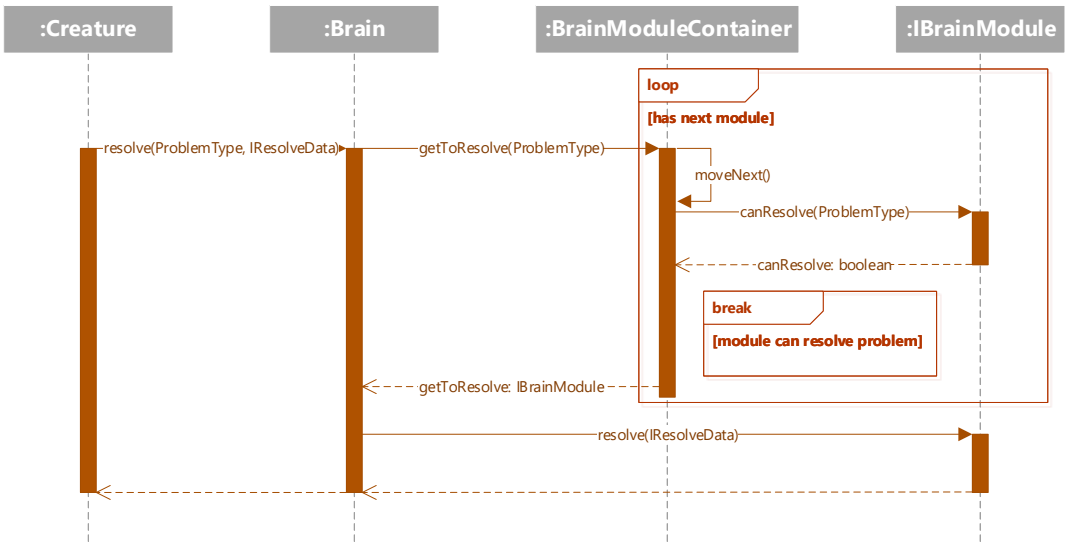
Pro účely řešení různých problémů, které mohou NPC potkat, bude každé NPC mít vlastní mozek, který bude složen z několika modulů, kde každý z modulů řeší konkrétní problém. Pokud NPC se má například rozhodnout, zda je pro něj cizí NPC nebezpečné na základě získaných zkušeností, předá vstupní data svému mozku, který data zpracuje a zvolí vhodné rozhodnutí. Celý systém AI je vyjádřen třídním diagramem na obrázku 48.

Pro vytváření instancí třídy *Brain* je vytvořena pomocná třída *BrainFactory*, která slouží k tomu, aby nebylo potřeba pro každé NPC složitě vytvářet specifickou instanci třídy, ale vytvoří se již přednastavená instance, jelikož každý mozek může obsahovat různé moduly, které řeší různé problémy. Ne všechny NPC musejí mít mozek obsahující všechny možné moduly, ale jen ty, které potřebuje pro řešení svých problémů. Každá třída *Brain* obsahuje kontejner pro moduly, třída *BrainModuleContainer*, do kterého se registrují implementace rozhraní *IBrainModule*, kde každý modul má přiřazenou množinu problémů, které modul řeší. Pokud NPC potřebuje řešit nějaký problém, zavolá na své instanci třídy *Brain* metodu *resolve*, která prohlédá kontejner modulů a najde modul, který daný problém řeší, následně mu předá data a nechá modul problém vyřešit a následně NPC vrátí řešení problému. Celý proces řešení problému znázorňuje sekvenční diagram na obrázku 49.

Pro jednotlivé moduly vždy existuje vlastní implementace rozhraní *IResolveData*, pomocí kterého NPC předává modulu informace potřebné k vyřešení problému. Interní řešení problémů



Obrázek 48: Třídní diagram modulu umělé inteligence



Obrázek 49: Sekvenční diagram procesy řešení problému pomocí mozku NPC

už mozek NPC nemusí řešit, celý mechanismus je postaven tak, že mozek může mít zaregistrované různé moduly pro různé problémy, přičemž každé NPC může stejný problém řešit pomocí jiného modulu. V rámci implementace byly vytvořeny moduly pro řešení chůze, rozhodování, zda je NPC nebezpečné, rozhodnutí, zda je pravděpodobné, že na NPC zaútočí jiné NPC, a také rozhodování, jaké úkoly NPC nabídne jinému NPC, jelikož některé NPC nemusí být schopno úkoly plnit, a proto na základě zkušeností je vhodné odhadovat pravděpodobnost, že NPC úkol splní a podle toho mu jej nabídnout či nikoli. Jednotlivé moduly si zde podrobněji rozepíšeme:

- *FightOrFleeBrainModule* – modul slouží pro vyhodnocení nebezpečnosti nepřítele na základě jeho vlastností. Cílem modulu je odhadnout DPS (zranění za sekundu, z anglického damage per second) nepřítele a na jeho základě vyhodnotit jeho nebezpečnost. Při každém souboji si NPC uloží do kolekce zkušenosti s daným nepřítelem, jeho vlastnosti a výsledné DPS. Tím si NPC postupně sbírá instance datové sady, nad kterou nejprve provede předzpracování, výběr atributů, čištění datové sady od instancí s chybějícími hodnotami a následně vybere vhodný model ML pro řešení odhadu DPS. Modely ML, mezi kterými je pro odhad DPS vybíráno, jsou lineární regrese, k-nejbližších sousedů a SVM. Při vypnutí aplikace se veškeré získané zkušenosti ukládají na disk a jsou k dispozici i při dalším zapnutí aplikace.

- *PredictStartFightingBrainModule* – modul slouží pro odhad, zda nepřítel při přiblížení zaútočí nebo ne. Dle získaných zkušeností s potkáváním NPC a následných útocích se postupně vytváří datová sada, ze které se pak určuje pravděpodobnost útoku.

- *QuestsOfferBrainModule* – modul slouží k rozhodování, zda rozdávač úkolů nabídne NPC nebo hráči úkoly. Když NPC přijímá úkoly a následně je plní anebo neplní, vytváření se

zkušenosti, které následně slouží k rozhodování o příští nabídce úkolů. Důležité je zmínit, že modul nepracuje s odhadem čistě na základě zkušeností získaných přímo s konkrétním NPC. Veškeré zkušenosti jsou zobecněné pro DPS NPC a velikost populace zvířecího druhu. Do algoritmus ML tedy vstupuje hodnota DPS a velikost populace, nikoli konkrétní vlastnosti NPC. Model ML pracuje s DPS z toho důvodu, že se dá předpokládat, že silné NPC se pravděpodobněji daleko lépe probije světem a nebude mít příliš problémů při plnění úkolů.

- *WalkingBrainModule* – modul využívá politiky Q-learningu vytvořené v samostatné aplikaci pro učení chůze. Modul na vstupu přijímá pozici nohou NPC a určuje pohyb nohou v dalším kroku.

5.5 Použité modely a jejich výstupy

5.5.1 Učení chůze

Jak již bylo zmíněno dříve, pro učení chůze byl zvolen algoritmus Deep Q-Learning. Vstupem algoritmu jsou pozice nohou, kde každá noha se může vyskytovat ve stavu *1* – zakročená anebo *0* – vykročená. Na výstup algoritmus vrací číslo, které představuje akci s nejvyšší Q-hodnotou. Pro provedení akce na základě čísla získaného z výstupu algoritmu je nejprve potřeba číslo převést na bitové pole, ve kterém každý bit náleží jedné noze. Pokud máme například NPC, které má 2 končetiny, ze kterých je první v pozici 1 (zakročená) a druhá v pozici 0 (vykročená), vstupem algoritmu je pole [1, 0]. Na výstup algoritmus vrací číslo 1, které je po převedení na bitové pole ve tvaru [0 1]. Každá končetina, která má na svém indexu v bitovém poli číslo 1, je při provedení dalšího kroku vykročena, ostatní nohy zůstávají v původní poloze.

Pokud by model chůze byl složitější, než je v našem případě, bylo by velmi složité najít optimální politiku, která maximalizuje odměnu získanou při chůzi NPC. V takovém případě by nám právě algoritmus Deep Q-Learning velmi pomohl, jelikož by nám optimální politiku vyhledal, aniž bychom na ni museli složitějším analytickým způsobem přijít sami. V našem modelovém případě však potřebujeme ukázat, že je tento algoritmus dobrou volbou pro učení chůze, což dokážeme pouze tím způsobem, že optimální politiku budeme předem znát a ověříme, že ji algoritmus dokáže vyhledat. Z tohoto důvodu jsme si definovali optimální politiku následovně: pokud je končetina NPC ve stavu 1 (zakročená), pak je optimální akce vykročit tuto nohu směrem k cílové pozici, a pokud je končetina ve stavu 0 (vykročená), pak je optimální akce s nohou nehýbat.

Aby skutečně námi definovaná optimální politika maximalizovala odměnu získanou při chůzi, byl pro výpočet odměny stanoven následující vzorec:

$$odměna = ušlá\ vzdálenost \cdot \frac{počet\ správně\ vykročených\ končetin}{celkový\ počet\ končetin} \tag{41}$$

Tabulka 1: Konfigurace algoritmu Deep Q-Learning pro učení chůze

Atribut	**Hodnota**
Celkový počet kroků	10 000
Počet epoch	10
Počet kroků v jedné epoše	1 000
Minimální hodnota ε	0.001
Počet kroků pro snížení ε na minimum	5 000
Koeficient snížení	0.1
Škálování odměny	1

Ze vzorce je patrné, že pokud NPC provádí kroky přesně podle optimální politiky, pak je jeho odměna rovna ušlé vzdálenosti v posledním kroku, ale pokud NPC provádí kroky přesně opačné, není mu uznána ušlá vzdálenost a získaná odměna je nulová. Nutno ještě podotknout, že v politice je zohledněn případ, kdy jsou všechny nohy ve stavu 0 (vykročené), přičemž v takovém stavu by optimální akce byla nedělat žádný pohyb. Z toho důvodu je v tomto případě odměna vždy rovna ušlé vzdálenosti.

Při testování algoritmu bylo použito několik konfigurací, ze kterých nejlépe vycházela konfigurace zobrazena v tabulce 1. Celkový čas potřebný pro naučení algoritmu byl 132 minut. Výsledkem učení chůze je politika, díky které se NPC je schopno pohybovat daleko rychleji než při náhodném střídání nohou při chůzi. Rozdíl v rychlosti chůze před a po naučení algoritmu je znázorněn na obrázcích 50 a 51, na kterých je zaznamenána chůze v průběhu 30 sekund. Obrázky byly pořízeny v trénovací aplikaci pohledem ze shora ve dvousekundových intervalech, přičemž chůze směřovala shora dolů. Učení bylo provedeno pro NPC se čtyřmi končetinami a výsledná politika je zobrazena v tabulce 2. Z tabulky lze vyčíst, že hodnoty akcí v binární podobě jsou shodné s hodnotami stavů končetin, jen jsou uvedeny v opačném pořadí. Tak je tomu správně, jelikož stavy končetin jsou do algoritmu vloženy v pořadí od první končetiny do poslední končetiny, přičemž nejméně významný bit, který představuje akci pro první končetinu, je v bitovém poli umístěn na pravé straně a nejvíce významný bit, který představuje akci pro poslední končetinu, je umístěn na levé straně. Výjimkou je první řádek, ve kterém jsou všechny končetiny ve stavu 0 (vykročená) a tedy ve kterém je vždy odměna rovna ušlé vzdálenosti v posledním kroku, a proto není možné z hlediska námi vymyšlené optimální politiky vyhodnotit, zda je akce správná či nikoli.

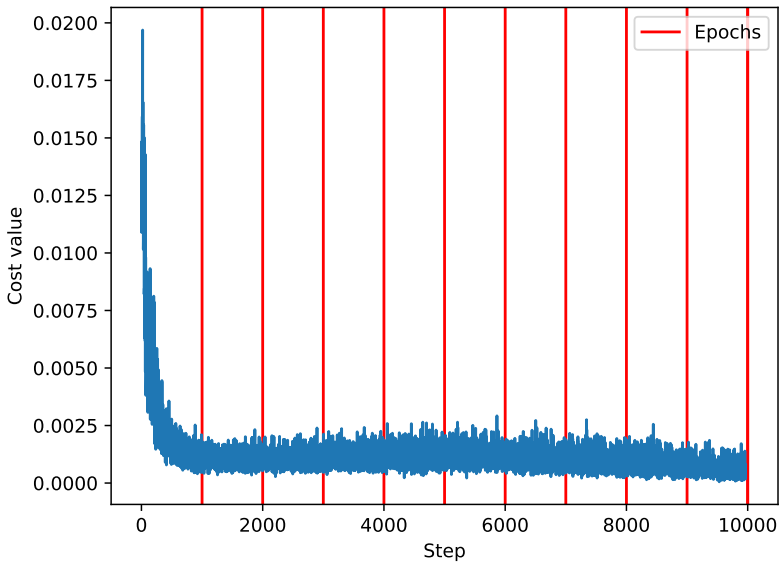
Hodnoty chybové funkce neuronové sítě v průběhu učení jsou zobrazeny na obrázku 52. Stojí za povšimnutí, že chybová funkce není v každém kroku snižována, jako bychom očekávali u klasického učení s učitelem, ale kolísá nahoru a dolů, přičemž má sestupnou tendenci až při celkovém zhodnocení. To je dáno parametrem ε, který nám zajišťuje to, že algoritmus nevybírá vždy akci s nejvyšší Q-hodnotou, ale s určitou pravděpodobností vybírá akci náhodnou. Až když dojde ke snížení hodnoty parametru ε na minimum, chybová funkce se ustálí a je snižována v každém kroku.

Tabulka 2: Výsledná politika získaná algoritmem Deep Q-Learning při učení chůze

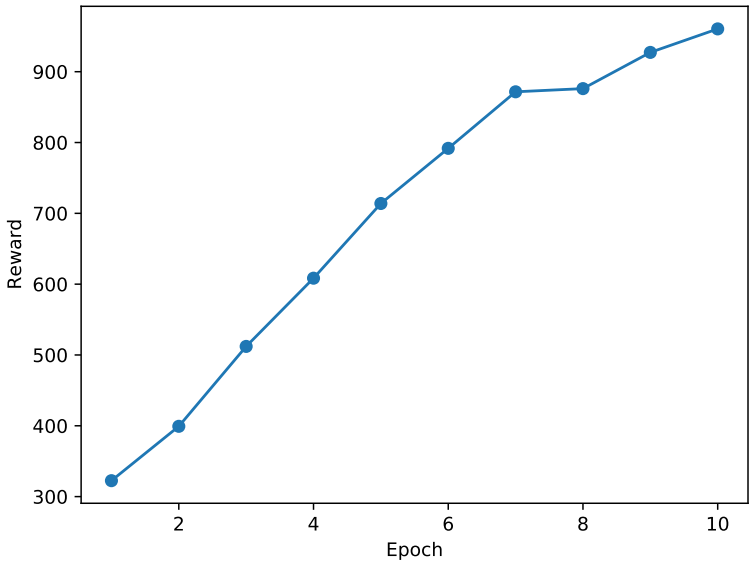
Stav končetin (1 - zakročená, 0 - vykročená)	**Akce** (binárně)
0, 0, 0, 0	1 1 1 1
0, 0, 0, 1	1 0 0 0
0, 0, 1, 0	0 1 0 0
0, 0, 1, 1	1 1 0 0
0, 1, 0, 0	0 0 1 0
0, 1, 0, 1	1 0 1 0
0, 1, 1, 0	0 1 1 0
0, 1, 1, 1	1 1 1 0
1, 0, 0, 0	0 0 0 1
1, 0, 0, 1	1 0 0 1
1, 0, 1, 0	0 1 0 1
1, 0, 1, 1	1 1 0 1
1, 1, 0, 0	0 0 1 1
1, 1, 0, 1	1 0 1 1
1, 1, 1, 0	0 1 1 1
1, 1, 1, 1	1 1 1 1

Obrázek 50: Rychlost chůze před naučením algoritmu Deep Q-Learning

Obrázek 51: Rychlost chůze po naučení algoritmu Deep Q-Learning



Obrázek 52: Hodnota chybové funkce v průběhu učení chůze



Obrázek 53: Vývoj průměrné odměny v průběhu učení chůze

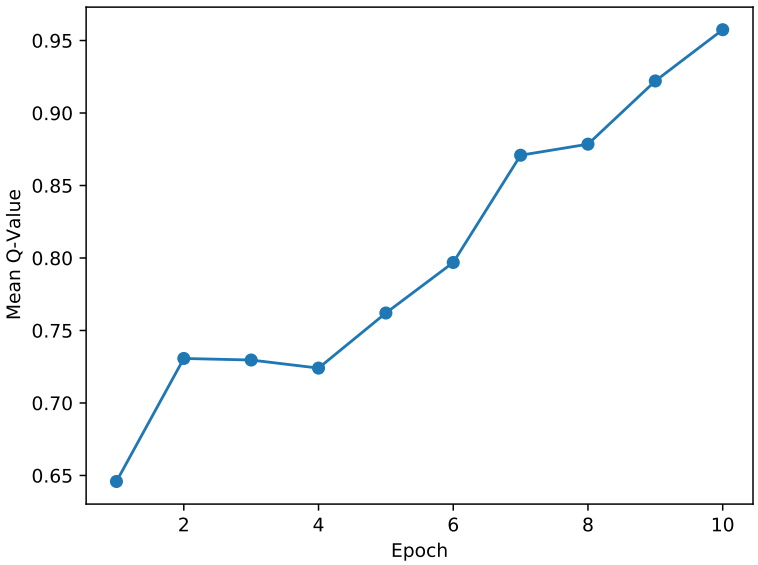
Kromě chybové funkce je výstupem učení také historie získaných odměn, která je zobrazena na obrázku 53, a také průměrná Q-hodnota za jednotlivé epochy znázorněna na obrázku 54.

Výsledek učení chůze hodnotím velmi dobře. Ukázali jsme si, že posilované učení má v tomto typu úloh rozhodně co nabídnout, přičemž není potřeba složitě připravovat data pro učení, ale stačí mít funkční prostředí, ve kterém se může agent pohybovat, a vhodně zvolenou funkci pro určování odměny. Pokud algoritmus běží dostatečně dlouho, pak je velmi pravděpodobné, že k vyhledání optimální, případě alespoň skoro optimální, politiky nakonec dojde.

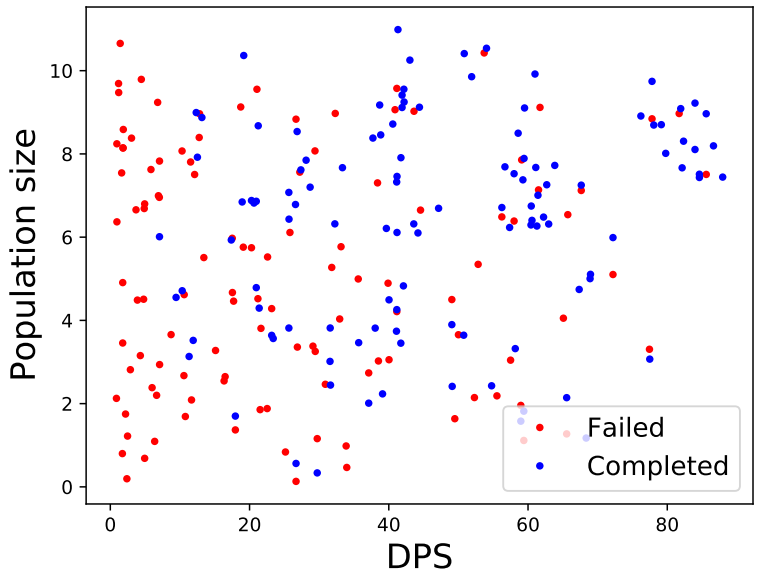
5.5.2 Adaptivní nabídka úkolů

Pro adaptivní nabídku úkolů nabízené rozdávači úkolů okolním NPC a hráči byla nejprve posbírána datová sada, kdy rozdávač úkolů nabízel úkoly všem bez ohledu na schopnost NPC plnit úkoly. Po nasbírání dostatečného množství zkušeností došlo k analýze problému a výběru mezi dvěma modely ML pro jeho řešení. Použitými modely byly logistická regrese a SVM. Cílem modelu bylo provést klasifikaci, jestli NPC spadá do třídy NPC s pravděpodobným selháním anebo do třídy s pravděpodobným dokončením úkolu. Pro klasifikaci byly použity 2 atributy, zranění udělené NPC za sekundu (DPS) a velikost populace daného zvířecího druhu NPC. Pomocí logistické regrese jsme docílili přesnosti 68.7% a pomocí SVM jsme docílili přesnosti 68.3%.

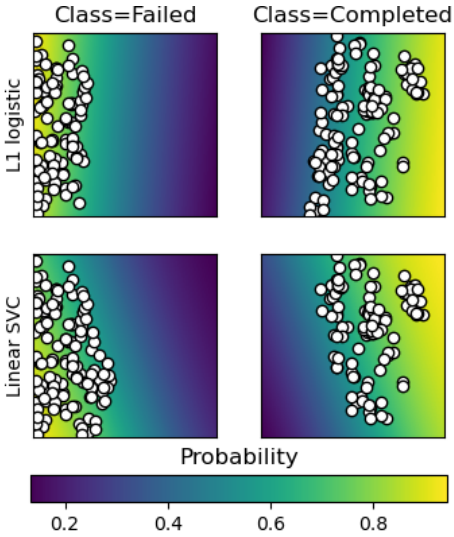
Datová sada použita pro trénování klasifikátorů je zobrazena na obrázku 55. Z pohledu na trénovací datovou sadu je patrné, že problém pro určení pravděpodobnosti splnění úkolů je komplexnější, než jsme si zadefinovali, čemuž odpovídá i dosažená přesnost. Jinými slovy řečeno, jednotlivé instance různých tříd jsou ne příliš dobře odděleny, což znamená to, že pouze atributy



Obrázek 54: Vývoj průměrné Q-hodnoty v průběhu učení chůze



Obrázek 55: Datová sada pro klasifikaci plnění úkolů



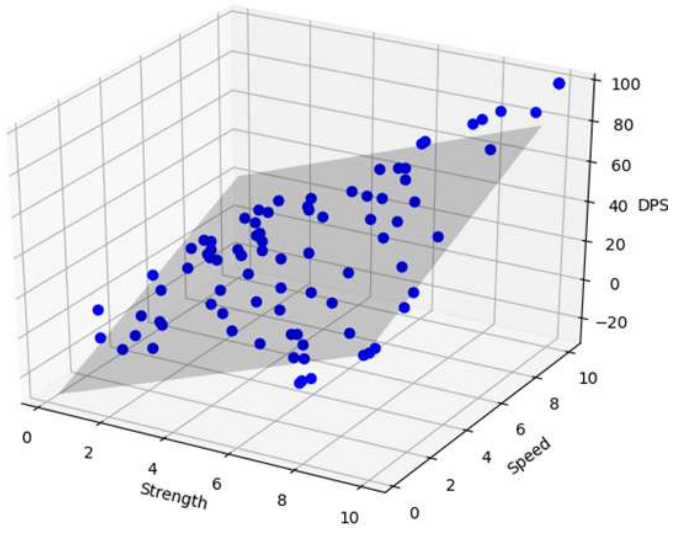
Obrázek 56: Pravděpodobnost splnění úkolů získaná klasifikátory

DPS a velikost populace nejsou dostatečné pro provedení přesnější klasifikace. Řešením tohoto problému by bylo přidání dalších atributů do datové sady, které by nám umožnily třídy lépe separovat. Pro naše účely nám však tato přesnost stačí. Výsledná klasifikace se znázorněním pravděpodobnosti pro výběr tříd je vidět na obrázku 56. Adaptivní nabídka úkolů způsobuje realistické chování rozdávačů úkolů, jelikož se rozhodují o nabídce úkolů pro NPC v závislosti čistě na získaných zkušenostech s plněním rozdaných úkolů.

5.5.3 Odhadování nebezpečnosti NPC

Tento problém je úzce spjat se soubojovým systémem a vlastnostmi NPC. Jak již bylo dříve napsáno, každé NPC má sadu vlastností, které určují jeho DPS. Cílem každého NPC je odhadovat nebezpečnost ostatních NPC tím, že odhadnou jejich DPS v závislosti na jejich vlastnostech, a podle toho se rozhodnout, zda mají na NPC zaútočit anebo od něj utéct. DPS je počítáno vzorcem: $DPS = Strength \cdot Speed$. Problém je řešen modulem *FightOrFleeBrainModule*. Modul řeší předzpracování dat, odstraní nízko korelující vlastnosti s výsledným DPS a následně vybírá mezi ML modely dle dosažené přesnosti po natrénování. Z vlastností jsou po předzpracování vybrány korektně atributy síla a rychlost. Problém je řešitelný jednoduchými modely, příkladem je lineární regrese zobrazena na obrázku 57.

Na první pohled se tento problém zdá být nesmyslné řešit, jelikož pokud bychom chtěli, aby NPC odhadovalo DPS ostatních NPC ne příliš přesně, stačilo by do vzorečku pro výpočet přidat šum, čímž by se okamžitě celkový dojem zlepšil, jelikož by NPC občas útočilo i na NPC, které nemůže v souboji porazit. Cílem implementace ML na tento problém je ukázat, že NPC může reálně vidět vlastnosti jiných NPC a na základě nich se rozhodovat, aniž by znalo přesný vzorec pro DPS. Tak ve skutečnosti funguje i reálný svět. Zvířata vidí na ostatních jejich drápy, zuby,



Obrázek 57: Odhadování DPS pomocí lineární regrese

silné tělo a podle toho zhodnotí, zda se jedná o nebezpečné zvíře. Při získávání zkušeností si zvíře vytvoří intuici pro odhadování nebezpečnosti ostatních zvířat. Pointou použití tohoto přístupu je to, že NPC ví o ostatních pouze to, co by reálně mohlo vědět i ve skutečném světě. NPC tedy zná vlastnosti, které jsou na ostatních viditelné, ale neví nic o ostatních výpočtech a modelech.

Pokud bychom uvažovali simulaci, ve které by míra zranění nebyla určena takto jednoduchým vzorcem, jako je to v našem případě, ale byla by odsimulována dle fyziky implementované v této simulaci, pak by se každé NPC skládalo z částí těla, které mají definovanou váhu, rozměr a další vlastnosti, a na základě těchto vlastností by při úderu některé z končetin do těla cizího NPC došlo k jeho poškození. Pochopitelně by tento proces byl výkonnostně daleko náročnější, než je tomu dnes v počítačových hrách. Na druhou stranu by tento přístup znamenal posun počítačových her směrem k simulacím, což by v mnohých ohledech znamenalo i větší realističnost a také by to mohlo vést k širším možnostem využití počítačových her.

6 Závěr

Cílem této práce bylo rozšířit projekt The Other Kosmos o modul AI pro zajištění chování NPC s důrazem na využití modelů ML za účelem zvýšení realističnosti simulovaného světa, s čímž souvisela i potřeba nastudovat oblast ML a přijít s možnostmi využití ML v počítačových hrách. V rámci práce byl projekt rozšířen o procedurálně generované bytosti, byla vytvořena aplikace pro trénování chůze těchto procedurálně generovaných bytostí, byl implementován soubojový systém a výpočet zranění při souboji mezi NPC, dále byl vytvořen adaptivní systém úkolů, které si NPC rozdávají mezi sebou a také je nabízejí hráči, a nakonec bylo implementováno chování NPC, které jim zvyšuje šanci na přežití díky sbírání zkušeností a odhadování nebezpečnosti okolních NPC.

Dle mého názoru byly všechny cíle práce splněny s tím, že výstupem práce jsou spíše nápady a přednesený přístup, s kterým by bylo možné dále pokračovat ve využívání ML v počítačových hrách a zajistit tak ve hrách větší realističnost.

Práce nejprve popisuje dnes používané algoritmy a způsoby implementace AI v počítačových hrách, následně čtenáře seznamuje s oblastí ML a nakonec ukazuje na několika příkladech v projektu The Other Kosmos, jakým způsobem by integrace ML do počítačových her mohlo probíhat a jaké typy problémů by mohlo ML ve hrách řešit.

V práci bylo uvažováno použití ML pouze pro chování NPC, přičemž reálné využití je daleko širší. V počítačových hrách by se například mohlo ML využít pro generování terénu anebo na vytvoření adaptivní obtížnosti.

Celkový přínos práce vnímám spíše jako zajímavý nápad na odklon od zaběhlých dogmat pro implementaci AI v počítačových hrách. V herním vývoji, stejně jako i v jiných odvětvích, bude vždy hrát velkou roli výsledná ekonomika věci. Riskovat neúspěch a finance investorů v už tak dost rizikovém a drahém průmyslu je možná až moc bláznivé.

Mě osobně však práce přinesla mnoho pozitivního. Seznámil jsem se s mnoha modely ML, mohl jsem si vyzkoušet ML na unikátních problémech a rozšířit si znalosti z programování při řešení neustálých problémů vznikajících při vývoji modulu do projektu The Other Kosmos.

Literatura

1. RUSSELL, Stuart J.; NORVIG, Peter. *Artificial intelligence a modern approach*. Prentice Hall, 2010.
2. BOURG, David M.; SEEMANN, Glenn. *Al for game developers: creating intelligent behavior in games*. OReilly, 2004.
3. MILLINGTON, Ian; FUNGE, John David. *Artificial intelligence for games*. Morgan Kaufmann/Elsevier, 2009.
4. RABIN, Steve. *Game AI pro: collected wisdom of game AI professionals*. CRC Press., 2014.
5. AURÉLIEN, Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. OReilly, 2019.
6. JAMES, Gareth; WITTEN, Daniela; HASTIE, Trevor; TIBSHIRANI, Robert. *An introduction to statistical learning: with applications in R*. Springer, 2017.
7. VOLNÁ, Eva. *Evoluční algoritmy a neuronové sítě* [online]. 2012 [cit. 2020-05-09]. Dostupné z: `https://www1.osu.cz/~volna/Evolucni_algoritmy_a_neuronove_site.pdf`.
8. PATTERSON, Josh; GIBSON, Adam. *Deep learning: a practitioner's approach*. OReilly, 2017.
9. BALAKRISHNAN, Suryakumar. *Learning Libgdx Game Development*. Packt Publishing, 2015.
10. *SQLite Home Page* [online]. 2000 [cit. 2020-05-10]. Dostupné z: `https://www.sqlite.org/index.html`.
11. GMBH, Data Geekery. *The easiest way to write SQL in Java* [online]. [N.d.] [cit. 2020-05-10]. Dostupné z: `https://www.jooq.org/`.
12. *Weka 3 - Data Mining with Open Source Machine Learning Software in Java* [online]. [N.d.] [cit. 2020-05-10]. Dostupné z: `https://www.cs.waikato.ac.nz/ml/weka/index.html`.
13. *Deep Learning for Java* [online]. [N.d.] [cit. 2020-05-10]. Dostupné z: `https://deeplearning4j.org/`.
14. ČERNÍN, Karel; DANĚK, Tomáš. *Teorie RPG* [online]. [N.d.] [cit. 2020-05-10]. Dostupné z: `http://www.taria.unas.cz/files/TeorieRPG.pdf`.

A Zdrojový kód metody *processAction*

```
public void processAction(int action) {
    feetConstraintsResetTime = 0f;
    actionProcessed = false;

    Matrix4 bodyTransform = new Matrix4();
    centerBody.getMotionState().getWorldTransform(bodyTransform);
    bodyTransform.getTranslation(previousBodyPosition);

    int numOfDiffBits = 0;
    double[] stateArray = lastState.toArray();
    if (lastState.getState() != 0) {
        for (int i = 0; i < bodyParts.size && i < Integer.SIZE; i++) {
            if (((action >> i) & 1) != (int)stateArray[i]) {
                numOfDiffBits++;
            }
        }
    }

    for (int i = 0; i < bodyParts.size && i < Integer.SIZE; i++) {
        int mask = 1 << i;

        if ((action & mask) != 0) {

            Matrix4 feetTransform = new Matrix4();
            Vector3 feetTranslation = new Vector3();
            bodyParts.get(i).getMotionState().getWorldTransform(feetTransform);
            feetTransform.getTranslation(feetTranslation);
            Vector3 loc = new Vector3(feetTranslation);
            Vector3 trans = new Vector3(feetTranslation);
            Vector3 move = new Vector3(movementTarget.x, 0, movementTarget.z);
            loc.sub(move);
            loc.nor();
            loc.scl((float)(bodyParts.size - numOfDiffBits) / (float)bodyParts.
                size);
            loc = trans.sub(loc);
            loc.y = 1;
            feetConstraints.get(i).setPivotB(loc);
```

```
        } else {
            Matrix4 feetTransform = new Matrix4();
            Vector3 feetTranslation = new Vector3();
            bodyParts.get(i).getMotionState().getWorldTransform(feetTransform);
            feetTransform.getTranslation(feetTranslation);
            Vector3 trans = new Vector3(feetTranslation);
            trans.y = 1;
            feetConstraints.get(i).setPivotB(trans);
        }
    }
    actionProcessingTime = actionProcessingTime % 1;
}
```

Výpis 3: Metoda pro provedení kroku NPC

B Zdrojový kód metody *generateSpeciesBody*

```
public static ArrayList<BodyPartModel> generateSpeciesBody() {
   ArrayList<BodyPartModel> result = new ArrayList<>();

   result.add(new BodyPartModel(
            0,
            BodyPartType.HEAD.getValue(),
            false,
            0,
            0,
            null
         ));

   int bodySize = Utils.randomRange(2, 10);

   for (int i = 1; i <= bodySize; i++) {
      result.add(new BodyPartModel(
            0,
            BodyPartType.BODY.getValue(),
            false,
            0,
            i,
            null
         ));

      if (i % 2 == 1) {
         result.add(new BodyPartModel(
               0,
               BodyPartType.LEG.getValue(),
               false,
               1,
               i,
               null
            ));

         result.add(new BodyPartModel(
               0,
               BodyPartType.LEG.getValue(),
```

```
                true,
                2,
                i,
                null
            ));

        result.add(new BodyPartModel(
                0,
                BodyPartType.LEG.getValue(),
                false,
                -1,
                i,
                null
            ));

        result.add(new BodyPartModel(
                0,
                BodyPartType.LEG.getValue(),
                true,
                -2,
                i,
                null
            ));
      }
   }

   return result;
}
```

Výpis 4: Generování těla NPC

C Zdrojový kód metody *generateSpecies*

```
public static SpeciesModel generateSpecies() {
   // name
   Nomen nameGenerator = Nomen.est().withCasing(Casing.CAPITALIZE).
       withSeparator(" ");
   if (Utils.random(1) == 1) {
      nameGenerator.adjective();
   }
   if (Utils.random(1) == 1) {
      nameGenerator.animal();
   }
   if (Utils.random(1) == 1) {
      nameGenerator.superb();
   }
   String name = nameGenerator.get();
   if (name.isEmpty()) {
      name = Nomen.est().person().get();
   }

   // species
   SpeciesModel species = new SpeciesModel(
         null,
         name,
         null,
         null,
         Utils.randomRange(SPECIES_POPULATION_SIZE_MIN,
             SPECIES_POPULATION_SIZE_MAX),
         false,
         null);

   // RPG stats
   species.statSet = new StatSetModel(
         null,
         Utils.randomRange(RPG_HP_MIN, RPG_HP_MAX),
         Precision.round(Utils.randomRange(RPG_STRENGTH_MIN, RPG_STRENGTH_MAX),
             2),
         Precision.round(Utils.randomRange(RPG_SPEED_MIN, RPG_SPEED_MAX), 2),
```

```
            Precision.round(Utils.randomRange(RPG_STAMINA_MIN, RPG_STAMINA_MAX), 2)
                ,
            Precision.round(Utils.randomRange(RPG_INTELLIGENCE_MIN,
                RPG_INTELLIGENCE_MAX), 2));

    // body
    species.bodyParts = CreatureBodyGenerator.generateSpeciesBody();

    return species;
}
```

Výpis 5: Generování zvířecího druhu